UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMATICA
Departamento de Sistemas Informáticos y Computación

TESIS DOCTORAL

Sobre la equivalencia entre semánticas operacionales y denotacionales para lenguajes funcionales paralelos

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Lidia Sánchez Gil

Directores

Mercedes Hidalgo Herrero
Yolanda Ortega Mallén

Madrid, 2015

Sobre la equivalencia entre semánticas operacionales y denotacionales para lenguajes funcionales paralelos

TESIS DOCTORAL

Memoria presentada para optar al grado de Doctor
Presentada por
Lidia Sánchez Gil

Dirigida por las doctoras
Mercedes Hidalgo Herrero
Yolanda Ortega Mallén

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

2015

On the equivalence of operational and denotational semantics for parallel functional languages

PhD Thesis

Lidia Sánchez Gil

Advisors
Mercedes Hidalgo Herrero
Yolanda Ortega Mallén

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

2015

Agradecimientos

Tengo tanto que agradecer y a tantas personas que podría rellenar páginas y páginas y aún me faltaría espacio. A lo largo de estos años han sido muchos los que me han ayudado y es imposible nombrar a todos. Así que he tenido que hacer una criba y quedarme con los más significativos.

En primer lugar quiero dar las gracias a Carlos, por estar ahí día a día, por aguantarme, por ocuparse de la casa y el peque para que yo pueda terminar con lo que ya empecé. Le conocí antes de empezar este proyecto y durante este tiempo ha vivido los altibajos de este largo camino. Con él he compartido los buenos momentos y él me ha hecho ver la otra cara de la moneda en los no tan buenos, pero siempre respetando mis decisiones. Me ha escuchado mil y una vez mientras yo pensaba en alto para ver si así conseguía aclarar mis ideas. Ha llevado con una sonrisa que no le contestara por estar pensando en mis cosas. Gracias por ser un marido fabuloso sin el cuál yo no estaría aquí. Y gracias a Ezequiel que, sin saberlo, ha puesto su granito de arena prescindiendo de mami muchos días y muchas noches.

También tengo que dar las gracias a esas maravillosas personas que me trajeron al mundo, mis papis, Luis y Rosa. Les agradezco todo lo que han hecho y siguen haciendo por mí. Por sus esfuerzos, a veces grandes, a veces pequeños, para darnos una educación y enseñarnos la suerte que tenemos de poder disfrutar de ella. Por su apoyo económico y el no económico que es, sin duda alguna, mucho mayor que el primero.

Y como esto va de familia, pues no puedo dejarme a mis hermanas. A Lucía por tenerla cerquita, a ella, a Juan Carlos y a Carlos y Rosa, por hacerme pasar buenos ratos cuando estamos juntos y suavizar los momentos de tensión disfrutando de buena compañía. A Ignicia por la lejanía, por mostrarme que el estar lejos no es sinónimo de olvido o de no querer. A ella y a todas las hermanas de Belén que he conocido en estos años les doy las gracias por su interés y por sus oraciones.

También agradezco a mi familia política su ayuda. En especial a mis suegros, Concha y Gonzalo, por venir a cuidar a Ezequiel mientras Carlos y yo curramos, y a Coty, la peque de la *family* por esas charlas interminables, por compartir conmigo un montón de risas y por hacer reír a Ezequiel como nadie.

Y aunque los agradecimientos a la familia podrían extenderse tanto cómo extensa es en sí misma, pasaré al grupo de amigos. Primero agradecerle a Mónica su amistad desde la infancia. Crecimos juntas y nos fuimos formando a la par, con nuestras vidas paralelas, tan distintas y tan iguales a la vez. Teníamos claro desde el principio lo que queríamos y por lo que luchar y bueno, quizá no todos, pero parte de esos sueños se han ido cumpliendo. Pese a que lo que llamamos la vida de adulto no nos permite vernos tanto como quisiéramos siempre, siempre, ha estado a golpe de teléfono y dispuesta a escucharme y quedar para invitarme a zumito de naranja. Y el pack Mónica-Lidia, se queda cojo sin el gran Ignacio

(Nacho para todos menos para nosotras). Hemos sido como hermanos, incluso llegamos a inventarnos nuestros propios apellidos para pasar como tales en los campamentos de la infancia, y a día de hoy sigue ahí, alegrándose de mis logros pese a querer usurparle la *Señora Cucharita*. Y a ambos os agradezco gran parte de lo que soy, pese a llamarme "bruja".

A mis queridas Peñita (y Juan) y Gemita, también tengo que agradecerles que haya llegado hasta aquí. Muchas horas de estudio compartidas y mucho apoyo y ánimos durante todo este tiempo. A mis dos grupos de mamás, en particular a Isa, Leti, Gema, Cami y Belén (Carla) por aguantarme cuando entro en pánico, por prestarme vuestro apoyo y por ayudarme a buscar sinónimos que encajen en el texto. A todo ese gran montón de amigos que han rezado por mí y que no voy a nombrar por ser una lista demasiado grande.

Y pasamos a las personas de ciencia. Aliseta, que podría encajar perfectamente en el grupo de amigos de antaño pero ha terminado estando en este lado. Gracias por esos enormes desayunos con zumo de naranja, café y barrita de jamón serrano a la plancha con queso fundido, complementados en ocasiones con un donuts de azúcar y otro de chocolate, que me sacaban de los bloqueos mentales y alimentaban mi cuerpo y mi espíritu. Gracias por tus consejos y por calmarme en los momentos de estrés. Otro de los personajes importantes de esta historia ha sido el gran, fabuloso y magnífico Ignacio Fábregas (no me odies por esto). Le conocía desde hacía bastante pero Marktdoberdorf me descubrió la gran persona que es y todo lo que sabe. Es mi pequeño dios, sacándome de apuros cada dos por tres, está siempre dispuesto a ayudar, y no sólo eso, es que ayuda de verdad. Me ha ayudado con la ciencia, con el dichoso LaTeX, y con un montón de charlas amenizadoras. Con David *Chico* he tratado menos que con Ignacio pero ha sido un compañero de despacho fabuloso y sí, también saca de apuros. David de Frutos es como un padre de la ciencia, si tienes una duda sabes que puedes recurrir a él, sabe de todo y siempre tiene un dato importante, un artículo interesante o unos comentarios que ayudan a perfeccionar lo hecho. Aunque en menor medida también han dejado su marca personas como Luis Llana, Maria Inés, Jorge Carmona, Alberto E. y Alberto V., mis compañeros de Máster Manuel Montenegro y Carlos Romero y otros compañeros de escuelas y congresos como Gaby, Nacho, Enrique, Adrián (que también me ha prestado gran ayuda en temas de papeleo), Henrique y Castro. También quiero agradecer los comentarios y consejos de Rita Loogen, Phil Trinder, Joachim Breitner y Arthur Charguéraud.

Yolanda y Mercedes, dos grandes mujeres y dos grandes directoras. Mucho ha sido el tiempo que me han dedicado. A veces, lo han tenido que sacar del tiempo dedicado a sus familias, por lo que este agradecimiento se hace extensible a Luis, Fernando, Jorge, Daniel y la pequeña Ana. Les doy las gracias por haber creído en este sueño incluso más que yo misma, por todo lo que me han enseñado tanto en lo referente a ciencia como en lo referente a la vida, por haberme guiado durante todo este largo camino, por haber conseguido de mí lo que no hubiera hecho jamás por mí misma.

Y aquí va el último agradecimiento, aunque me atrevería a decir que el más importante de todos. Todas las personas mencionadas anteriormente (y muchas de las no mencionadas) han contribuido a que yo esté aquí. Pero como casi todo doctorando hay momentos en los que uno quiere tirar la toalla y dedicarse a otros menesteres, y no soy menos que los demás, pero en mi caso podría decir que llegué a tirar la toalla. Muchos trataron de convencerme y de que siguiera adelante, pero mi decisión estaba tomada. Entré en el despacho de Narciso Martí Oliet a decirle que todo se terminaba y, aún no sé como, salí con la determinación de que terminaría la tesis. No ha pasado ni un solo día en el que no le haya agradecido sus palabras, que seguirán en mi interior como han estado todo este tiempo. Solamente puedo decir GRACIAS.

Para concluir he de decir que mi tesis ha formado parte de los proyectos de investigación: StrongSoft (TIN2012-39391-C04-04) financiado por el Ministerio de Economía y Competitividad, PROMETIDOS (S2009/TIC-1465) financiado por la Comunidad de Madrid, DESAFIOS10 (TIN2009-14599-C03-01) financiado por el Ministerio de Ciencia e Innovación, DESAFIOS (TIN2006-15660-C02-01) financiado por el Ministerio de Educación y Ciencia, WEST(TIN2006-15578-C02-01) financiado por el Ministerio de Educación y Ciencia, PROMESAS (ref. S-0505/TIC-0407) financiado por la Comunidad de Madrid y la ayuda predoctoral FPI (BES-2007-16823) financiada por el Ministerio de Educación y Ciencia.

Resumen

Tal y como se indica en [ede14], `Eden` es un lenguaje funcional paralelo que extiende `Haskell` con construcciones sintácticas para especificar la creación de procesos. Como explican los autores de [BLOP96], en `Eden` se distinguen dos partes: un λ-cálculo perezoso y expresiones de coordinación. El lenguaje `Jauja` es una simplificación de `Eden` que mantiene sus principales características. El objetivo de esta tesis es dar los primeros pasos para demostrar la equivalencia entre las semánticas definidas para `Jauja` por Hidalgo-Herrero en [Hid04]. Se quiere probar la equivalencia en términos de *corrección* y *adecuación computacional* entre una semántica operacional y una semántica denotacional. Para hacerlo nos basamos en las ideas expuestas por Launchbury en [Lau93], en el que se demuestra la equivalencia entre una semántica natural y una semántica denotacional estándar para un λ-cálculo extendido con declaraciones locales.

Puesto que demostrar la equivalencia entre las semánticas definidas para Jauja supone un estudio demasiado complejo para afrontarlo en un primer paso, hemos comenzado por considerar una extensión del lenguaje utilizado por Launchbury al que se ha añadido una aplicación paralela que da lugar a creaciones de procesos y comunicaciones entre ellos, es decir, a un sistema distribuido formado por distintos procesos que interactúan entre sí. A partir de este sencillo lenguaje el estudio se desarrolla en varias etapas en las que se establece la equivalencia entre distintas semánticas operacionales y denotacionales para modelos distribuidos y no distribuidos. La semántica operacional del modelo distribuido heredada de `Jauja` es una semántica de paso corto para varios procesadores. Para realizar la equivalencia de esta semántica con una semántica denotacional estándar extendida, con objeto de dotar de significado a la aplicación paralela, se introducen dos semánticas intermedias: una de paso corto pero limitada a un único procesador y una semántica de paso largo que es una extensión de la semántica natural de Launchbury. En el caso de prescindir de las aplicaciones paralelas, la semántica natural de Launchbury y nuestra extensión se comportan igual. Con respecto al modelo no distribuido, y con el fin de completar las demostraciones ausentes en el trabajo de Launchbury, se construye un espacio de funciones para los valores de la semántica denotacional con recursos introducida por el autor. Posteriormente, se comprueba que es equivalente a la semántica denotacional estándar bajo la condición de disponer de infinitos recursos. También se estudian algunas relaciones existentes entre *heaps* y pares (*heap*, término) que se aplican para estudiar la equivalencia de las dos semánticas operacionales introducidas por Launchbury.

Hemos realizado gran parte del estudio utilizando la *notación localmente sin nombres*, situada a medio camino entre la de nombres y la de de Bruijn. Así se evitan los problemas derivados de la notación con nombres, es decir, tener que trabajar con términos α-equivalentes. Por otra parte, también se eluden las desventajas de utilizar solo los índices de de Bruijn, que resultan complicados de manejar y dificultan la lectura de los términos.

Abstract

The programming language `Eden` [ede14] is a parallel functional language that extends `Haskell` with some syntactic constructs for explicit process specification and creation. `Eden` [BLOP96] comprises two differentiated parts: A lazy λ-calculus and coordination expressions. The programming language `Jauja` is a simplification of `Eden` that gathers its main characteristics. The target of this thesis is to give the first steps in the proof of the equivalence between the semantics defined for `Jauja` by Hidalgo-Herrero in [Hid04]. We prove the equivalence in terms of *correctness* and *computational adequacy* of an operational semantics with respect to a denotational one. We base our work on Launchbury's ideas that are introduced in [Lau93], where he proved the equivalence between a natural semantics and a standard denotational semantics for a λ-calculus extended with local declarations.

Since the study of the equivalence between the semantics defined for `Jauja` is too complex, we start with the study of the language used by Launchbury extended with a parallel application. This new expression gives rise to the creation of processes and the communication between them, i.e., to a distributed model with several processes. The study is developed in several steps, with different operational and denotational semantics for distributed and non-distributed models.

The operational semantics of the distributed model inherited from `Jauja` is a small-step semantics for several processors. In order to prove the equivalence between this semantics and an extension of the standard denotational semantics, we introduce two intermediate semantics: A small-step semantic restricted to one processor, and an extension of Launchbury's natural semantics. When no parallel application is involved, Launchbury's extension and the original natural semantics have the same behavior.

The study of the non-distributed model leads to the construction of an appropriate function space for the values of the resourced denotational semantics introduced by Launchbury. This resourced semantics and the standard denotational one are equivalent when infinitely many resources are provided. We also define a preorder relation on heaps, that is extended to (heap, term) pairs. We use this preorder to establish a relation between the heaps and values produced when the same (heap, term) pair is evaluated with different semantics.

We use the *locally nameless representation*, which is halfway between the named notation and the de Bruijn notation. This alternative avoids the problems derived from the named representation, i.e., dealing with α-equivalence, as well as the disadvantages of using only indices.

Índice general

Índice de figuras

Parte I

Resumen de la Investigación

Capítulo 1

¿Qué, por qué y cómo?

Cuando un lenguaje de programación es dotado de varias semánticas, estas semánticas tienen que ser equivalentes, es decir, asociar significados equivalentes a cada programa escrito en el lenguaje. En la tesis doctoral de Hidalgo Herrero [Hid04], la autora define una semántica operacional y una semántica denotacional para `Jauja`, una simplificación de `Eden` [BLOP96] en la que se distinguen dos partes:

- un λ-cálculo perezoso;
- expresiones de coordinación.

En aquel trabajo, la equivalencia entre ambas semánticas quedó como problema abierto. Por eso, el objetivo inicial de esta tesis fue abordar el estudio de las relaciones existentes entre dichas semánticas. Comenzamos basándonos en las ideas expuestas por Launchbury en [Lau93] para demostrar la equivalencia entre una semántica natural de paso largo y una semántica denotacional estándar de un λ-cálculo extendido con declaraciones locales. Sin embargo, `Jauja` y el lenguaje tratado por Launchbury son considerablemente distintos. El primero, tal y como se ha expuesto anteriormente, está compuesto no sólo por las expresiones inherentes a un λ-cálculo perezoso, sino que consta además de expresiones de coordinación. Afrontar de golpe la equivalencia entre las semánticas de `Jauja` no parecía viable. Por ello, el objetivo final de esta tesis no es probar dicha equivalencia, sino iniciar el proceso para conseguirlo.

Comenzamos incorporando al λ-cálculo de Launchbury una aplicación paralela que dará lugar a creaciones de procesos y comunicaciones entre ellos. Esto conlleva tener que extender las semánticas previamente definidas para dar significado a la nueva expresión y a los nuevos identificadores, que ahora representan variables y canales. Estas extensiones han de ser coherentes tanto con las definiciones de Launchbury como con las de Hidalgo Herrero.

En la tesis hay dos partes diferenciadas: por un lado, un modelo distribuido formado por distintos procesos que interactúan entre sí y, por otro, un modelo más sencillo con un único procesador.

Paradójicamente, el estudio del modelo distribuido fue el que dio lugar a que se profundizara más en el modelo con un único procesador. La parte referente a un solo procesador se divide a su vez en dos secciones: en la primera se trabaja con semánticas denotacionales, estudiando también sus características y la relación entre ellas; en la segunda se estudia el funcionamiento y las propiedades de distintas semánticas operacionales y las relaciones existentes entre ellas. Ha sido necesario profundizar en la teoría de dominios para definir correctamente el espacio de algunos de los valores semánticos con los que se trabaja y poder establecer las relaciones entre distintos espacios.

Como era de esperar, no han sido pocos los problemas que se han encontrado a lo largo del estudio, y a los que hemos tenido que dar solución. Parte del trabajo realizado ha sido consecuencia de la ausencia de demostraciones detalladas para los resultados que propuso Launchbury en [Lau93]. En dicho trabajo se exponen las ideas intuitivas sobre las que se han de construir las demostraciones, pero el desarrollo de las mismas es bastante más complicado de lo que se muestra en dicho artículo y lo que en un primer momento parece resolverse con una simple inducción por reglas ha resultado ser mucho más complejo. Dado que diversos trabajos [BKT00, HO02, NH09, Ses97, vEdM07] se basan en este estudio de Launchbury, se ha considerado de gran importancia formalizar los resultados expuestos en él.

Por otra parte, la notación con la que se representan las expresiones de los lenguajes puede facilitar o dificultar las demostraciones formales. En el caso del λ-cálculo, es bastante frecuente encontrar problemas relacionados con los nombres elegidos para expresar un término, es decir, problemas derivados de la α-conversión. Se han desarrollado distintas técnicas para evitarlos, como por ejemplo la notación de de Bruijn [dB72], la representación localmente sin nombres [Cha11], o las técnicas de la lógica nominal [Pit13]. En nuestro caso se ha elegido la segunda opción, con la que hemos trabajado en algunos de los artículos que componen esta tesis.

En resumen, la búsqueda de la equivalencia entre las semánticas de un modelo distribuido nos ha hecho adentrarnos y profundizar en distintas semánticas para un modelo más sencillo y en diversas técnicas derivadas de notaciones alternativas para expresar los términos del lenguaje.

1.1 Objetivos de la tesis

El objetivo principal de la tesis ha sido:

- encauzar la demostración de la equivalencia entre las semánticas definidas para `Jauja` en [Hid04].

Dicho propósito ha quedado desglosado en los siguientes objetivos específicos:

- extender el λ-cálculo con una aplicación paralela, es decir, incluir en el lenguaje un operador para introducir explícitamente el paralelismo;
- definir para este λ-cálculo extendido distintos modelos semánticos con uno y con varios procesadores, tanto operacionales como denotacionales;
- estudiar las relaciones entre los modelos semánticos definidos: formalizar la equivalencia entre las semánticas definidas en el paso anterior;
- formalizar algunas de las demostraciones ausentes en [Lau93]: en concreto la equivalencia entre una semántica denotacional estándar y una de recursos y la equivalencia entre la semántica natural definida por Launchbury y su versión alternativa.

1.2 Organización de la tesis

Esta tesis se presenta como una colección de publicaciones ya realizadas. Para entender la relación entre los artículos de esta colección y obtener una visión de conjunto, se ha completado el trabajo con este capítulo introductorio y tres capítulos más que se enumeran a continuación: en el Capítulo 2 se explican los conceptos previos que se consideran

necesarios para poder entender el estudio realizado. El Capítulo 3 está dedicado a los resultados obtenidos. Más que detallar cada uno de ellos, lo que se ha hecho en las distintas publicaciones, este capítulo pretende dar una idea intuitiva de ellos de forma que facilite la lectura y comprensión de los artículos. Cada sección del capítulo está ligada a una o varias publicaciones que se indican explícitamente. También en este capítulo se enumeran y comentan algunos de los trabajos de otros autores relacionados con esta tesis. Por su parte, el trabajo futuro se desarrolla en el Capítulo 4. Está dividido en cuatro secciones y en ellas se indica si ya se ha realizado una parte de ese trabajo. Finalmente, el Capítulo 5 recoge las cuatro publicaciones principales que componen esta tesis:

- P1: *Relating function spaces to resourced function spaces* [SGHHOM11].
- P2: *A Locally Nameless Representation for a Natural Semantics for Lazy Evaluation* [SGHHOM12b].
- P3: *The Role of Indirections in Lazy Natural Semantics* [SGHHOM14b].
- P4: *An Operational Semantics for Distributed Lazy Evaluation* [SGHHOM10].

Además se han incluido dos apéndices. El Apéndice A contiene las versiones extendidas de las publicaciones P2 y P3. En dichas extensiones se detallan todas las demostraciones realizadas para obtener los resultados expuestos en las publicaciones.

- TR1: *A locally nameless representation for a natural semantics for lazy evaluation (extended version)* [SGHHOM12c].
- TR2: *The role of indirections in lazy natural semantics (extended version)* [SGHHOM13].

Finalmente, el Apéndice B está formado por dos trabajos presentados en su momento como trabajo en progreso y cuyo desarrollo se ha postergado por diversas razones:

- WP1: *Launchbury's semantics revisited: On the equivalence of context-heap semantics* [SGHHOM14a].
- WP2: *A formalization in Coq of Launchbury's natural semantics for lazy evaluation* [SGHHOM12a].

Capítulo 2

¿Qué estaba hecho?

En este capítulo se repasan algunos conceptos que consideramos necesarios para entender la investigación realizada en esta tesis.

2.1 Lenguajes de programación

Todo lenguaje lleva asociado una sintaxis y una semántica. Según la *Real Academia Española* [Esp14], la sintaxis es "el conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje", mientras que la semántica es "el estudio del significado de los signos lingüísticos y de sus combinaciones, desde un punto de vista sincrónico o diacrónico" (según aparece en el avance de la vigésima tercera edición). De manera informal podemos decir que la sintaxis muestra cómo construir correctamente expresiones y la semántica dota de significado a esos términos bien construidos. Esto también es aplicable a los lenguajes de programación, donde la sintaxis indica cómo construir programas y la semántica indica cómo se comportarán esos programas al ser ejecutados en una computadora.

A veces, en los lenguajes naturales encontramos oraciones cuyo significado no es único. Por ejemplo: *Ana cogió su bicicleta*. Esta frase es ambigua, pues si Ana está jugando con Pablo en el parque no sabemos si ha cogido la bicicleta de Pablo o su propia bicicleta. Sin embargo, los lenguajes de programación vienen dotados de semánticas formales que impiden la ambigüedad de sus significados: a cada término le corresponde un único significado.

2.1.1 Lenguajes de programación funcionales

Existen diversos paradigmas de programación. En el imperativo el programador define paso a paso la solución a un problema, alejándose de la definición matemática inicial. Por contra, los lenguajes funcionales elevan el nivel de abstracción.

Consideremos por ejemplo cómo calcular la potencia n-ésima de un número. Su definición matemática podría ser la siguiente:

$$x^0 = 1$$
$$x^{n+1} = x \cdot x^n$$

En un lenguaje imperativo la expresión para la función potencia pierde su similitud

con la definición anterior, como puede verse al implementarla en `C`:

```
int potencia(int x, int n);
{
    int i = 1;
    int resultado = 1;
    while (i <= n)
    {
        resultado  = resultado * x;
        i++;
    }
    return(resultado);
}
```

Mientras que en un lenguaje funcional como `Haskell`, la implementación quedaría:

```
potencia x 0  =  1
potencia x n  = x  *  potencia x (n - 1)
```

En muchas ocasiones se atribuye a los lenguajes funcionales una menor eficiencia en su ejecución. En la edición revisada de [BB00], Barendregt y Barendsen explican que la arquitectura Von-Neumann se basa en la máquina de Turing. Los lenguajes de programación imperativos siguen una secuencia de instrucciones que se ajusta a dicha arquitectura. Sin embargo, son las máquinas de reducción las que se diseñan para la ejecución de lenguajes funcionales basados en el λ-cálculo. La mayor parte de las computadoras actuales tienen una arquitectura Von-Neumann, dando lugar ciertamente a una menor eficiencia de los lenguajes funcionales. Por ese motivo, gran parte de la investigación sobre lenguajes de programación funcional se ha dedicado a la implementación eficiente de estos lenguajes, habiéndose alcanzado hoy en día unas velocidades de ejecución verdaderamente competitivas. Esto, unido a las muchas ventajas que ofrecen los lenguajes funcionales desde el punto de vista del desarrollador (mayor nivel de abstracción, código más reducido, ausencia de efectos colaterales —transparencia referencial—, fácil depuración de programas, concurrencia, actualizaciones "en caliente" —*hot code deployment*—, recursión natural, etc), ha hecho florecer la programación funcional en ámbitos distintos del académico [OSV10]. Por ejemplo, `Haskell` es utilizado por algunas empresas como Intel y algunos bancos como Deutsche Bank; y ciertas partes de Facebook o Google están programadas usando este lenguaje (una lista más detallada puede encontrarse en [has14a]). Erlang también es utilizado en la industria [erl14a], por ejemplo en Whatsapp [wha14], Facebook y T-Mobile [erl14b].

2.1.2 Estrategias de evaluación

Las expresiones de los lenguajes de programación funcionales se evalúan mediante la reducción de subexpresiones. Dependiendo del orden de reducción establecido para los *redexes* (expresiones de reducción) se obtienen diferentes estrategias de evaluación que pueden encuadrarse en dos grandes grupos: el primero está formado por las estrategias en las que la evaluación de los argumentos se realiza antes de aplicar la función aunque no sean requeridos (evaluación impaciente); el segundo, por aquellas en las que los argumentos sólo se calculan cuando se necesitan sus resultados (evaluación perezosa).

Existen numerosas estrategias de evaluación, aunque aquí sólo se detallan tres de ellas, por ser las que aparecen en el desarrollo de esta tesis: *call-by-value*, *call-by-name* y *call-by-need*. Según las definiciones de Reade en [Rea89].

- *Call-by-value*: es una estrategia de evaluación impaciente en la que los argumentos son evaluados por completo antes que el cuerpo de la función;
- *Call-by-name*: en este caso el argumento (sin evaluar) es sustituido en el cuerpo de la función y la expresión resultante es evaluada, tratándose, por tanto, de una estrategia del segundo grupo. De esta forma es posible que algunas expresiones sean evaluadas más de una vez, aunque si no son requeridas no se evaluarán nunca;
- *Call-by-need*: es una estrategia de evaluación perezosa más eficiente que la estrategia anterior, ya que una vez obtenido el valor de una expresión este se guarda y comparte, y así no debe ser calculado de nuevo.

2.1.3 Lenguajes funcionales paralelos

La proliferación de máquinas paralelas y distribuidas hace que surja la necesidad de diseñar lenguajes que faciliten la programación paralela, y los lenguajes funcionales paralelos ofrecen grandes ventajas para ello. Si bien los lenguajes imperativos son eficientes, tratan a un nivel de abstracción muy bajo conceptos clave como la sincronización y la comunicación. Sin embargo, los lenguajes funcionales son una buena opción debido a su alto nivel de abstracción, a la transparencia referencial y a su modelo semántico claro (ventajas que ya han sido comentadas en la Sección 2.1.1).

Loogen realiza una clasificación del paralelismo en lenguajes funcionales en [Loo99] distinguiendo tres grandes grupos, dependiendo de la libertad que se deje al programador para establecer los puntos del programa susceptibles de ser evaluados en paralelo:

- *Paralelismo implícito*: es el inherente a la semántica de reducción, donde los *redexes* independientes pueden ser reducidos en un orden arbitrario o en paralelo. Es la base de la paralelización automática de los lenguajes funcionales.
- *Paralelismo semi-explícito*: el programador indica dónde desearía una evaluación en paralelo añadiendo anotaciones para el compilador. Bien se utilizan construcciones paralelas de alto nivel como esqueletos [Col89], bien estrategias de evaluación [THLP98]. Pero estas anotaciones podrían ser ignoradas por el compilador.
- *Paralelismo explícito*: el programador establece dónde computar distintas expresiones en paralelo. Existen extensiones de algunos lenguajes de programación como `Haskell` [Pey03] o ML [MTH90] con construcciones para la creación explícita de procesos, la comunicación de valores y la sincronización entre procesos.

El lenguaje funcional `Haskell` [Pey03, has14b] ha sido la base de numerosas versiones paralelas y distribuidas, como se señala en [TLP03]. La evaluación en `Haskell` es perezosa (Sección 2.1.2). Este tipo de evaluación restringe la explotación del paralelismo, pues las expresiones sólo se evalúan bajo demanda. Por eso las versiones paralelas de `Haskell` tratan de eliminar la pereza, ya sea mediante el trabajo especulativo, permitiendo la evaluación de partes no demandadas (como por ejemplo en `GpH` [THLP98] con el operador `par`), o bien introduciendo estrictez, al forzar la evaluación de partes antes de que su resultado sea necesario (el operador `seq` en `GpH` [THLP98]).

2.1.4 El lenguaje funcional paralelo `Eden`

El lenguaje que ha inspirado los trabajos de esta tesis es `Eden` [BLOP96, LOP05, ede14], una extensión de `Haskell` con construcciones de coordinación para controlar la evaluación

en paralelo. La coordinación en `Eden` se basa en la definición explícita de procesos y en la comunicación implícita mediante *streams*. A continuación, se resumen las principales características de `Eden` (según se indica en [Hid04]):

- *Abstracciones de proceso*: son las expresiones que de un modo puramente funcional definen el comportamiento general de un proceso.
- *Creaciones de proceso*: son aplicaciones de las anteriores a un grupo determinado de expresiones que conformarán los valores de los canales de entrada del nuevo proceso creado.
- *Comunicaciones entre procesos*: son asíncronas e implícitas, pues el paso de mensajes no lo ha de explicitar el programador. Además, estas comunicaciones no tienen por qué ser de un único valor, sino que pueden realizarse en forma de *streams*.

Además, las construcciones de `Eden` se extienden para modelizar sistemas reactivos:

- *Creación dinámica de canales*: sin esta facilidad las comunicaciones son jerárquicas entre procesos padre y procesos hijo. Pero los canales dinámicos permiten romper esta jerarquía, permitiendo topologías de comunicación más complejas.
- *No-determinismo*: para poder modelizar las comunicaciones de varios a uno, se introduce la abstracción de proceso que toma varios *streams* devolviendo uno sólo que es una mezcla no determinista de los elementos de los anteriores.

2.2 Semánticas de lenguajes de programación

En el prefacio del texto de Winskel [Win93] se explica que dotar de una semántica formal a un lenguaje de programación consiste en construir un modelo matemático. Las semánticas formales permiten comprender y razonar sorbre el comportamiento de los programas.

2.2.1 Semánticas formales

Dependiendo del uso que se le quiera dar, se considerará un tipo de semántica formal u otro. Destacamos aquí los dos utilizados en esta tesis:

- **Operacional:** la semántica operacional de un lenguaje describe el significado de un programa especificando cómo se ejecuta en una máquina abstracta. Esta semántica se centra en conocer el resultado que genera un programa y el modo en que éste es obtenido. Distinguimos dos categorías: las *semánticas de paso corto*, que describen cómo se realiza cada computación paso a paso; y las *semánticas de paso largo, o naturales*, que describen cómo se obtiene directamente el resultado final.
- **Denotacional:** la semántica denotacional dota de significado a los programas construyendo unos objetos matemáticos, llamados *denotaciones*, que describen el significado de las expresiones del lenguaje. Podríamos decir que se trata de encontrar objetos matemáticos que representen lo que hace un programa. Una semántica denotacional viene dada por la función que computa el programa, pero no se ocupa de cómo se llega a ello. La denotación de un término se obtiene componiendo las denotaciones de sus subtérminos. Por tener un mayor nivel de abstracción que la semántica operacional, permite estudiar más fácilmente la equivalencia entre programas. La

forma usual de definir una semántica denotacional se centra en los siguientes aspectos: definir el espacio de significados; dotar a cada constante del lenguaje de un significado en dicho espacio; construir funciones semánticas sobre el espacio de significados para cada operador del lenguaje; y, finalmente, definir la función semántica principal que indica el valor semántico de cada programa.

Cuando hay más de un tipo de semántica definida para el mismo lenguaje, hay que demostrar que estas son equivalentes. En el caso de las semánticas operacionales y denotacionales, esta equivalencia suele darse en términos de corrección y adecuación computacional:

- **Corrección:** indica que las reducciones operacionales preservan el significado denotacional de los términos.
- **Adecuación:** la adecuación computacional de una semántica operacional con respecto a una denotacional establece que si una expresión está definida según la semántica denotacional, entonces existe una reducción operacional para ella.

En esta tesis se trabaja con distintas semánticas operacionales y denotacionales para un lenguaje de programación funcional, y se estudian las relaciones existentes entre ellas.

2.3 Espacios de funciones

En algunas ocasiones, comparar programas que están escritos en lenguajes de programación diferentes puede ser bastante complicado si se utilizan semánticas operacionales cuyas transiciones se construyen a partir de la sintaxis del lenguaje, tal y cómo se explica en [Win93]. Por ello surge la necesidad de dar significado a las expresiones de una forma más abstracta, mediante una semántica denotacional cuyos valores se encuentran en un espacio de funciones.

Abramsky y Jung en [AJ94] introducen los dos problemas que dan lugar a la teoría de dominios [Sco73]: el menor punto fijo como significado de definiciones recursivas y las ecuaciones de dominios recursivos. Asímismo, Abramsky en [Abr91] explica cómo la teoría de dominios, introducida por Scott, ha sido estudiada tanto desde el marco teórico como aplicado, en particular al campo de las semánticas denotacionales.

2.3.1 Conceptos básicos

A continuación, vamos a repasar algunos conceptos clave de la teoría de dominios. Daremos sus definiciones siguiendo el texto de Winskel sobre semánticas formales para lenguajes de programación [Win93].

Un conjunto P dotado de una operación binaria, $\sqsubseteq$, es un *orden parcial* si la relación es reflexiva, transitiva y antisimétrica.

Dado un subconjunto $X \subseteq P$, $p \in P$ es una *cota superior de* X si cualquier elemento de X es menor o igual que p, es decir, $\forall q \in X \;.\; q \sqsubseteq p$. Además, esta cota será mínima ($\bigsqcup X$) si cualquier otra cota superior es mayor que ella.

Un orden parcial $(P, \sqsubseteq)$ será completo (*cpo*) si para toda cadena infinita creciente de elementos $(d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots)$ existe una cota superior mínima ($\bigsqcup_n d_n$) en P. Si además está dotado de un elemento mínimo ($\bot$), se dirá que es un *orden parcial completo con mínimo*.

Dados dos *cpos* $(D, \sqsubseteq_D)$ y $(E, \sqsubseteq_E)$, una función $f : D \to E$ es *monótona* si $\forall d, d' \in D \;.\; d \sqsubseteq_D d' \;\Rightarrow\; f(d) \sqsubseteq_E f(d')$. Además, será *continua* si es monótona y para cada

cadena infinita ($d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$) se cumple que la cota superior mínima de las imágenes coincide con la imagen de la cota superior mínima de los elementos, es decir, $\bigsqcup_n f(d_n) = f(\bigsqcup_n d_n)$.

Dado un *cpo* $(D, \sqsubseteq_D)$ y una función continua $f : D \to D$, se dice que un elemento $d \in D$ es un *punto fijo de* f si $f(d) = d$.

Teorema de Kleene del punto fijo. *Sea* $(D, \sqsubseteq_D)$ *un* cpo *con mínimo y* $f : D \to D$ *una función continua. Se define* $\texttt{fix}(f) = \bigsqcup_n f^n(\bot)$. *Se verifica que*

1. $\texttt{fix}(\texttt{f})$ *es un punto fijo de* f*, es decir,* $f(\texttt{fix}(f)) = \texttt{fix}(f)$*;*
2. *Si* $f(d) = d$ *entonces* $\texttt{fix}(f) \sqsubseteq d$.

Luego $\texttt{fix}(\texttt{f})$ *es el menor punto fijo de* f.

Dados dos *cpos* $(D, \sqsubseteq_D)$ y $(E, \sqsubseteq_E)$, el *espacio de funciones* $[D \to E]$ consiste en los elementos $\{f \mid f : D \to E \text{ es continua}\}$ ordenados punto a punto mediante $f \sqsubseteq g \stackrel{\text{def}}{=} \forall d \in D. f(d) \sqsubseteq g(d)$. Esto hace que el espacio de funciones sea un *cpo* y para cada cadena infinita $f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_n \sqsubseteq \ldots$ la cota superior mínima cumple: $(\bigsqcup_n f_n)d = \bigsqcup_n (f_n(d))$.

2.3.2 Construcción de la solución inicial

El λ-cálculo puro y los lenguajes funcionales perezosos no se corresponden en su totalidad, ya que hay que distinguir entre elementos *convergentes*, aquellos cuya evaluación da lugar a funciones de D en D (siendo D el dominio adecuado de valores), y elementos *divergentes*, aquellos cuya evaluación no termina. Abramsky en [Abr90] hace referencia a este hecho y propone, para poder representar los elementos convergentes y divergentes, una teoría basada en *sistemas de transición aplicativos*, introduciendo la ecuación de dominios, $D = [D \to D]_\bot$, donde $[D \to D]_\bot$ corresponde al espacio de funciones continuas de D en D con el mínimo ($\bot$) añadido. Esta ecuación tiene una solución inicial no trivial que constituye un modelo para los lenguajes perezosos. La construcción de esta solución inicial viene detallada en [AO93] y aquí se hará un breve resumen de los pasos principales.

Sean D y E dos *cpos*. Se dice que $\langle i, j \rangle$ es un *embedding* de D en E si i y j son funciones continuas $D \stackrel{i}{\rightarrowtail} E \stackrel{j}{\twoheadrightarrow} D$ que verifican que $i \circ j \sqsubseteq \texttt{id}_E$ y $j \circ i = \texttt{id}_D$, donde $\stackrel{i}{\rightarrowtail}$ representa un inyección y $\stackrel{j}{\twoheadrightarrow}$ una proyección.

La construcción del espacio de funciones se realiza por niveles, que se definen de forma recursiva mediante $D_0 \stackrel{\texttt{def}}{=} \{\bot\}$ y $D_{n+1} \stackrel{\texttt{def}}{=} [D_n \to D_n]_\bot$. Para cada par de niveles consecutivos se pueden construir las funciones continuas $D_n \stackrel{i_n}{\rightarrowtail} D_{n+1} \stackrel{j_n}{\twoheadrightarrow} D_n$, donde $\langle i_n, j_n \rangle$ forman un *embedding*.

El primer nivel está formado por un dominio con un único elemento, tal y cómo indica la definición de D_0. El siguiente nivel estará formado por dos elementos, por un lado el elemento indefinido, $\bot_{D_1}$, y por otro la función continua de $\{\bot_{D_0}\}$ en $\{\bot_{D_0}\}$. A esta función la llamaremos d_1. En el tercer nivel se tienen cuatro elementos, uno corresponde al valor indefinido del nivel, $\bot_{D_2}$, y los otros tres a las funciones continuas de D_1 en D_1. Puesto que en D_1 hay dos elementos y verifican que $\bot_{D_1} \sqsubseteq d_1$, existen tres funciones continuas: d_{20}, d_{21} y d_{22} tales que $d_{20}(\bot_{D_1}) = \bot_{D_1} \sqsubseteq d_{20}(d_1) = \bot_{D_1}$, $d_{21}(\bot_{D_1}) = \bot_{D_1} \sqsubseteq d_{21}(d_1) = d_1$ y $d_{22}(\bot_{D_1}) = d_1 \sqsubseteq d_{22}(d_1) = d_1$, respectivamente. Estos tres niveles se representan en la Figura 2.1.

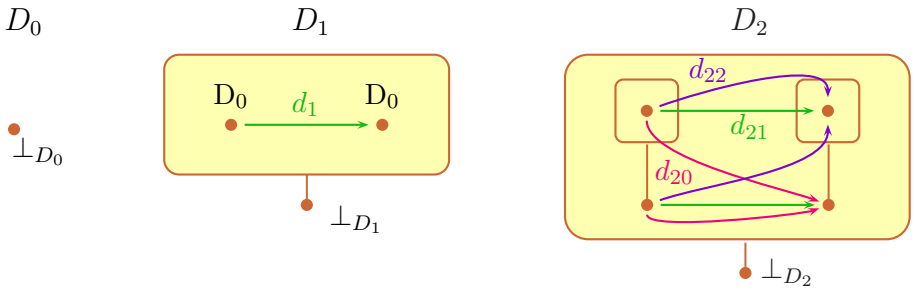


Figura 2.1: Primeros niveles del espacio de funciones $[D \to D]_\perp$

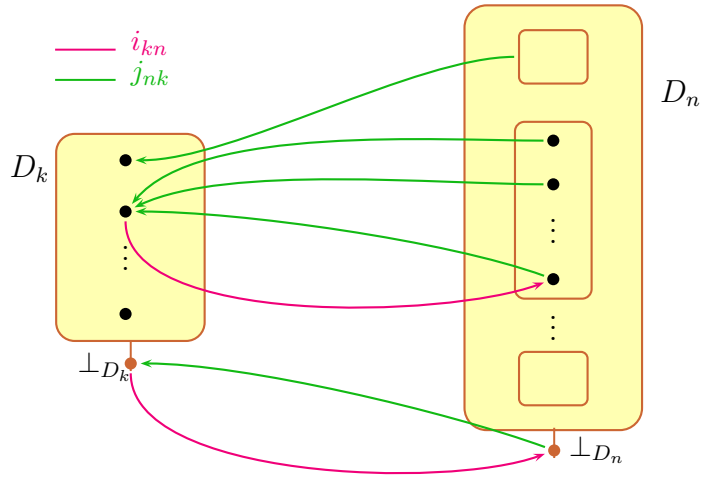


Figura 2.2: Inyecciones y proyecciones entre niveles

Existe una generalización de los *embeddings*, de forma que se puede pasar del nivel k al nivel n mediante la inyección i_{kn} y la proyección j_{nk}. Sin embargo, estas funciones no son exactamente inversas. Cuando pasamos de un nivel a otro superior mediante una inyección, se busca un valor de ese nivel cuya proyección corresponda al valor de inicio. Ahora bien, en ese nivel superior se dispone de más información, así que habrá más de un valor que cumpla el requisito; de entre todos ellos se elige el más indefinido. Por lo tanto, $i_{kn} \circ j_{nk} \sqsubseteq id_n$ y $j_{nk} \circ i_{kn} = id_k$, propiedad que viene heredada de los *embeddings* entre niveles consecutivos. La Figura 2.2 muestra esta situación para $n > k$.

Nótese que $\langle D_n,\ j_n\rangle_{n\in\omega}$ es un sistema inverso de cpo's. D está definido como el límite inverso del sistema anterior, es decir, $D = \lim_{\leftarrow}\langle D_n,\ j_n\rangle_{n\in\omega}$ y la solución inicial se identifica con $D = \{\langle x_n : n \in \omega\rangle : x_n \in D_n \wedge j_n(x_{n+1}) = x_n\}$. Se denota por ψ_n a la proyección $j_{\infty n} : D \twoheadrightarrow D_n$ y por ϕ_n a la inyección $i_{n\infty} : D_n \rightarrowtail D$. Tal y como explica Abramsky y Ong [AO93] se considera D_n como un subconjunto de D, es decir, si $x \in D_n$ entonces se identifica $\phi_n(x)$ con x, y si $x \in D$ entonces $\psi_n(x)$ se identifica con $x_n \in D_n$. Por lo que $D = \bigcup_n D_n$. Los valores denotacionales para el λ-cálculo están definidos sobre el dominio $D = [D \to D]_\perp$.

$$\frac{}{\lambda x.P \Downarrow \lambda x.P} \qquad \frac{M \Downarrow \lambda x.P \quad P[x := Q] \Downarrow N}{M\ Q \Downarrow N}$$

Figura 2.3: Relación binaria en Λ^0

2.3.3 Bisimulación Aplicativa

Para explicar el concepto de bisimulación aplicativa dado en [Abr90], se considera un λ-cálculo donde los λ-términos cerrados, representados por Λ^0, son considerados programas, y las λ-abstracciones valores. Se define una relación binaria $\Downarrow \subseteq \Lambda^0 \times \Lambda^0$, cuyas reglas se muestran en la Figura 2.3. Teniendo en cuenta esta relación se dirá que un término M *converge*, denotado por $M \Downarrow$, si existe algún término N tal que $M \Downarrow N$; en caso contrario se dirá que M *diverge*. Es decir, un término o bien converge a una λ-abstracción, o bien diverge.

Esta relación es la base para definir la *bisimulación aplicativa*. Tal y como indican Abramsky y Ong en [AO93], tendremos que determinar si un término converge observándolo por etapas. Dado un término cerrado M, en la primera etapa sólo podemos observar si M converge a una abstracción $\lambda x.M_1$. Si es así, se observa si al dar como argumento a dicha función el término N_1, ésta converge, es decir, si $M_1[x := N_1]$ converge. Y así sucesivamente.

Se define sobre Λ^0 una secuencia de relaciones binarias $\langle \sqsubseteq^B_k : k \in \mathbb{N} \rangle$, de la siguiente forma:

- $\forall M, N\ .\ M \sqsubseteq^B_0 N$.
- $M \sqsubseteq^B_{k+1} N \stackrel{\texttt{def}}{=} M \Downarrow \lambda x.P \Rightarrow (N \Downarrow \lambda x.Q\ \wedge\ \forall R \in \Lambda^0 . P[x := R] \sqsubseteq^B_k Q[x := R])$.
- $M \sqsubseteq^B N \stackrel{\texttt{def}}{=} \forall k \in \mathbb{N}\ .\ M \sqsubseteq^B_k N$.

Nótese que en el nivel 0 todos los términos cerrados están relacionados. En el resto de niveles dos términos convergentes estarán relacionados si al aplicarles el mismo argumento están relacionados en el nivel anterior. Esto viene derivado del hecho de que sólo es observable la convergencia de términos; es decir, sólo puede observarse si un término reduce a una λ-abstracción, pero no se puede observar lo que hay dentro de ella, es decir, su cuerpo. Por eso, la única forma de "observar" el cuerpo de la λ-abstracción es estudiar el comportamiento de ésta al aplicarle un argumento. Finalmente, si dos términos están relacionados en cada uno de los niveles, se dirá que están relacionados.

2.4 Semántica natural para evaluación perezosa

Launchbury presentó en [Lau93] una semántica natural perezosa (*call-by-need*, ver Sección 2.1.2) que ha sido de gran importancia en el paradigma funcional. En el texto el autor explica que la pereza implica un lenguaje no estricto, que ciertas reducciones sean compartidas y que la evaluación termine al encontrar una λ-abstracción. Otros lenguajes no estrictos que se usan en la actualidad, son, por ejemplo, `Miranda` [mir15] o `Haskell`[has15]. El trabajo de Launchbury ha sido citado con frecuencia y ha servido como base para otros trabajos y extensiones [BKT00, HO02, NH09, Ses97, vEdM07]. El éxito de este trabajo radica en su simplicidad. Las expresiones se evalúan dentro de un contexto que se

$$
\begin{array}{lcll}
x & \in & \mathit{Var} & \\
e & \in & \mathit{Exp} & ::= \quad x \mid \lambda x.e \mid (e\ x) \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e
\end{array}
$$

Figura 2.4: Sintaxis restringida del λ-cálculo extendido

$$
\begin{array}{rcll}
(\lambda x.e)^* & = & \lambda x.(e^*) & \\
x^* & = & x & \\
(\texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e)^* & = & \texttt{let}\ \{x_i = (e_i^*)\}_{i=1}^{n}\ \texttt{in}\ (e^*) & \\
(e_1\ e_2)^* & = & \begin{cases} (e_1^*)\ e_2 & \text{si } e_2 \text{ es una variable} \\ \texttt{let}\ y = (e_2^*)\ \texttt{in}\ (e_1^*)\ y & \text{e.o.c.,} \\ & \text{siendo } y \text{ una variable fresca} \end{cases} &
\end{array}
$$

Figura 2.5: Normalización del λ-cálculo extendido

representa como un conjunto de pares (variable/expresión) donde toda la información es compartida. Además, estos pares se actualizan sustituyendo las expresiones por su valor una vez calculado. De esta forma se modeliza la evaluación perezosa.

Launchbury trabaja con un λ-cálculo extendido con declaraciones locales recursivas como muestra la Figura 2.4, en el que se aplica un proceso de normalización en dos pasos:

- En primer lugar, se realiza una α-conversión, de forma que todas las variables ligadas mediante las declaraciones locales y las λ-abstracciones se renombran con variables frescas. De este modo todas las variables locales tienen nombres distintos.
- En segundo lugar, se fuerza a que los argumentos de las funciones sean variables, tal como se muestra en la Figura 2.5. Este cambio se denota por e^*.

Este proceso de normalización simplifica considerablemente las definiciones de las reglas de la semántica operacional. Por un lado, el hecho de usar nombres distintos hace que el ámbito de aplicación sea irrelevante. Por otro, la restricción sobre las aplicaciones consigue que no haya que introducir clausuras nuevas en la semántica.

En la semántica natural con estrategia *call-by-need* que define Launchbury, los juicios o sentencias son de la forma

$$\Gamma : e \Downarrow \Delta : w,$$

es decir, se evalúa la expresión e en el contexto del heap Γ, que reduce a un valor w en el contexto del heap Δ. Los *heaps* son funciones parciales de variables a expresiones. Se denomina *ligadura* a un par (variable, expresión) y se denota por $x \mapsto e$. Los *valores* ($w \in \mathit{Val}$) son expresiones en forma normal débil de cabeza (*whnf*, del inglés *weak-head-normal-form*), es decir, con una λ en cabeza. Las reglas semánticas se muestran en la Figura 2.6. Durante la evaluación de una expresión, se pueden añadir al heap nuevas ligaduras (regla Let). Así mismo, algunas de las ya existentes pueden ser actualizadas con sus correspondientes valores ya calculados (regla Var). La regla Lam indica que las expresiones ya evaluadas se reducen a ellas mismas sin modificar el contexto de evaluación. A pesar de la normalización, en la regla Var es necesaria una α-conversión del valor final obtenido que viene representado por $\hat{w}$. Este renombramiento evita colisiones con los nombres ya

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \quad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : w}$$

Figura 2.6: Semántica natural

$$\begin{aligned}
[\![\lambda x.e]\!]_\rho &= F_n(\lambda\nu.[\![e]\!]_{\rho \sqcup \{x \mapsto \nu\}}) \\
[\![e\ x]\!]_\rho &= ([\![e]\!]_\rho)\downarrow_{F_n} ([\![x]\!]_\rho) \\
[\![x]\!]_\rho &= \rho(x) \\
[\![\texttt{let}\ x_1 = e_1 \ldots x_n = e_n\ \texttt{in}\ e]\!]_\rho &= [\![e]\!]_{\{\!\{x_1 \mapsto e_1 \ldots x_n \mapsto e_n\}\!\}_\rho}
\end{aligned}$$

Figura 2.7: Semántica denotacional

existentes y se justifica por la convención de variables de Barendregt [Bar84]. La regla App reduce primero el término e y tras obtener un valor (es decir, una λ-abstracción) realiza la aplicación mediante una β-reducción, evaluando la expresión resultante. Por último, la regla Let, además de introducir en el *heap* las declaraciones locales, evalúa el cuerpo de la expresión. Nótese que debido a la normalización realizada previamente no puede haber conflictos entre las variables cuando éstas son introducidas en el *heap*.

A su vez, Launchbury también dotó de significado denotacional a las expresiones del λ-cálculo basándose en el modelo de Abramsky [Abr90]. La función semántica de la que parte es la siguiente:

$$[\![-]\!] \ : \ \mathit{Exp} \to \mathit{Env} \to \mathit{Value}$$

donde *Exp* representa las expresiones del λ-cálculo (Figura 2.4), *Value* un dominio apropiado que satisface la ecuación $\mathit{Value} = [\mathit{Value} \to \mathit{Value}]_\perp$ (explicado en la Sección 2.3), y *Env* contiene los entornos de evaluación de las variables libres. Los entornos son funciones de variables a valores, es decir,

$$\rho \in \mathit{Env} = \mathit{Var} \to \mathit{Value}.$$

La función semántica se incluye en la Figura 2.7, donde se utiliza una función que relaciona los *heaps* con los entornos:

$$\{\!\{-\}\!\} \ : \ \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$$

Esta función captura la recursión generada por las declaraciones locales y viene definida por:

$$\{\!\{x_1 \mapsto e_1 \ldots x_n \mapsto e_n\}\!\}_\rho \ = \ \mu\rho'.\rho \sqcup (x_1 \mapsto [\![e_1]\!]_{\rho'} \ldots x_n \mapsto [\![e_n]\!]_{\rho'})$$

En esta definición el operador de menor punto fijo viene representado por μ. Esta función puede verse como un modificador de entornos que sólo cobra sentido si los entornos y los *heaps* son consistentes; es decir, siempre que una variable aparezca ligada tanto en el entorno como en el *heap*, entonces estará ligada a valores para los que exista una cota superior.

$$\textsc{Var}\quad \frac{(\Gamma, x \mapsto e) : \hat{e} \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow \Delta : w} \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \quad (\Theta, y \mapsto x) : e' \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

Figura 2.8: Semántica natural alternativa

$$\begin{aligned}
\mathcal{N}[\![e]\!]_\sigma\ \bot &= \bot \\
\mathcal{N}[\![\lambda x.e]\!]_\sigma\ (S\ k) &= F_n(\lambda\nu.\mathcal{N}[\![e]\!]_{\sigma \sqcup \{x \mapsto \nu\}}\ ) \\
\mathcal{N}[\![e\ x]\!]_\sigma\ (S\ k) &= (\mathcal{N}[\![e]\!]_\sigma\ k)\downarrow_{F_n}\ (\mathcal{N}[\![x]\!]_\sigma\ )\ k \\
\mathcal{N}[\![x]\!]_\sigma\ (S\ k) &= \sigma\ x\ k \\
\mathcal{N}[\![\texttt{let}\ x_1 = e_1 \ldots x_n = e_n\ \texttt{in}\ e]\!]_\sigma\ (S\ k) &= \mathcal{N}[\![e]\!]_{\mu\sigma'\ (\sigma \sqcup x_1 \mapsto \mathcal{N}[\![e_1]\!]_{\sigma'}\ \sqcup \cdots \sqcup x_n \mapsto \mathcal{N}[\![e_n]\!]_{\sigma'}\ )}\ k
\end{aligned}$$

Figura 2.9: Semántica denotacional con recursos.

Launchbury define un orden sobre los entornos de forma que $\rho \leq \rho'$ si ρ' liga más variables que ρ, pero las que estén en ambos entornos deben estar ligadas a los mismos valores. Formalmente, $\forall x \in \mathit{Var}\ .\ \rho(x)\ \neq \bot \Rightarrow\ \rho(x) = \rho'(x)$.

2.4.1 Propiedades

Launchbury establece la *corrección* (Sección 2.2.1) de las reglas operacionales con respecto a la semántica denotacional expuesta. El teorema de corrección afirma que las reducciones preservan el significado de los términos y solamente se modifica el significado de los *heaps* añadiendo nuevas ligaduras, si ello fuera necesario.

Teorema 1 (Corrección de la semántica natural.)
Si $\Gamma : e \Downarrow \Delta : z$ *entonces para todo entorno* ρ, $[\![e]\!]_{\{\!\{\Gamma\}\!\}_\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}_\rho}$ *y* $\{\!\{\Gamma\}\!\}_\rho \leq \{\!\{\Delta\}\!\}_\rho$.

Dado que existen ciertas diferencias entre la semántica operacional y la semántica denotacional definidas, Launchbury introduce dos nuevas semánticas más próximas entre sí para establecer la *adecuación computacional* (Sección 2.2.1). En primer lugar, modifica la semántica natural cambiando las reglas para la variable y la aplicación por las expuestas en la Figura 2.8. En esta versión de la semántica no hay actualización de ligaduras y la aplicación se realiza a través de indirecciones, en vez de mediante una β-reducción. Las nuevas reglas hacen que los contextos de evaluación se ajusten más a los entornos de la semántica denotacional.

En segundo lugar, introduce una semántica denotacional basada en recursos, en la que, si no se dispone de recursos suficientes, los términos quedan indefinidos. En esta versión con recursos la función semántica toma un nuevo argumento, los recursos, que se van consumiendo por cada nivel sintáctico evaluado. De esta forma se consigue que la semántica denotacional se ajuste más a la aplicación de las reglas de la semántica operacional. Las nuevas cláusulas denotacionales se muestran en la Figura 2.9.

Finalmente, Launchbury demuestra la adecuación computacional de la semántica operacional alternativa con respecto a la semántica denotacional de recursos.

Teorema 2 (Adecuación computacional de la semántica alternativa.)
Si existe $m \in \mathbb{N}$ *tal que* $\mathcal{N}[\![e]\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma\ \sqcup \cdots \sqcup x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma\ )}\ (S^m\ \bot) \neq \bot$, *entonces existen un* heap Δ *y un valor* w *tal que* $(x_1 \mapsto e_1\ \ldots\ x_n \mapsto e_n) : e \Downarrow \Delta : w$.

$$
\begin{array}{rcl}
E & ::= & x \mid \backslash x.E \mid E_1\ E_2 \mid E_1 \# E_2 \mid \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E \\
 & \mid & \texttt{new}(y,x)E \mid x\ !\ E_1\ \texttt{par}\ E_2 \mid E_1 \bowtie E_2 \mid \Lambda[x_1\ :\ x_2].E_1 [\![E_2 \mid L \\
L & ::= & \texttt{nil} \mid [E_1\ :\ E_2]
\end{array}
$$

Figura 2.10: Sintaxis de `Jauja`

2.5 El lenguaje `Jauja` y las semánticas formales de `Eden`

El lenguaje `Jauja` definido por Hidalgo-Herrero en [Hid04] es una simplificación del lenguaje funcional paralelo `Eden` (introducido en la Sección 2.1.4) que recoge sus principales características. Como ya se ha mencionado, está formado por dos partes diferenciadas: un λ-cálculo perezoso y sus expresiones de coordinación. Estas últimas permiten introducir paralelismo mediante la creación explícita de procesos que interaccionan entre sí a través de canales de comunicación. También incorporan no-determinismo y, por tanto, reactividad. En esta tesis se utiliza un subconjunto de este lenguaje.

La sintaxis de `Jauja` está expuesta en la Figura 2.10. Las primeras expresiones corresponden a las propias de un λ-cálculo con declaraciones locales a las que se ha añadido la creación de procesos #. Sin embargo, no es posible que se comuniquen los procesos hijos entre sí y para ello se incluye otra construcción, $\texttt{new}(y,x)E$, con la que se crean *canales dinámicos*. La *conexión dinámica*, $x\ !\ E_1\ \texttt{par}\ E_2$, conlleva la evaluación en paralelo de E_1 y E_2, y la comunicación del valor de E_1 a través de x. El no-determinismo explícito de `Eden` se integra en `Jauja` mediante la expresión $E_1 \bowtie E_2$, que mezclará los dos *streams* o listas obtenidos a partir de E_1 y E_2. La expresión $\Lambda[x_1\ :\ x_2].E_1 [\![E_2$, permite tratar con listas que pueden ser vacías, `nil`, o no vacías, $[E_1\ :\ E_2]$.

2.5.1 Semántica Operacional

Hidalgo-Herrero construye una semántica operacional para `Jauja` [Hid04] que modeliza sus características fundamentales: evaluación perezosa y paralelismo dentro de un proceso y entre procesos. Da lugar a un modelo distribuido en el que se distingue una estructura en dos niveles: por un lado se tiene un sistema distribuido S formado por procesos paralelos, considerado el nivel superior; por otro lado, cada uno de estos procesos se encuentra en el nivel inferior y viene representado por un *heap* de ligaduras, H_i, como se muestra en la Figura 2.11.

Este modelo distribuido en dos niveles queda reflejado en la semántica operacional definida en [Hid04], donde se distinguen dos tipos de reglas: las reglas locales, que expresan cómo evoluciona cada uno de los procesos de forma individual; y las reglas globales, que muestran cómo evoluciona el sistema, indicando cómo se crean nuevos procesos y cómo se comunican entre sí. A continuación se explican brevemente las reglas de ambos niveles.

Las reglas locales indican cómo evoluciona un *heap* etiquetado, es decir, una colección de ligaduras con etiquetas que muestran su estado: A si la ligadura está activa, B si está bloqueada, es decir, a la espera de la evaluación de otra ligadura, e I si está inactiva, es decir, o ya está evaluada o no ha sido demandada. Cada regla se centra en una ligadura activa y el proceso evoluciona según se indique. Por ejemplo, la regla local (app-demand)

$$H\ +\ \{x \overset{I}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\ y \ \longrightarrow\ H\ +\ \{x \overset{A}{\mapsto} E, \theta \overset{B}{\mapsto} x\ y\}$$

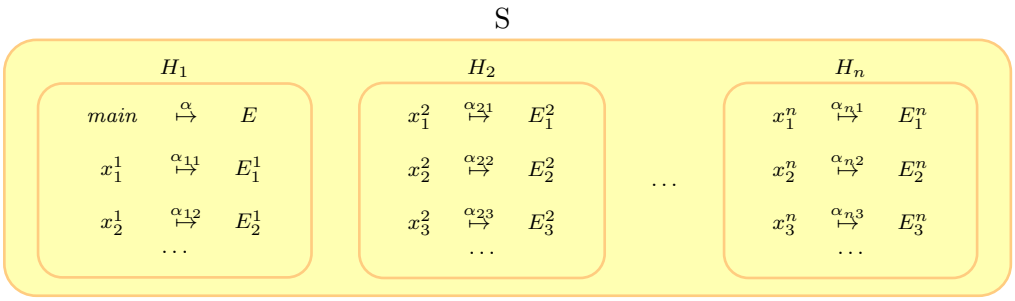


Figura 2.11: Modelo distribuido

expresa que al evaluar una aplicación hay que activar la ligadura referente al cuerpo de la aplicación y bloquear la ligadura demandante.

Las transiciones entre sistemas vienen dadas por $\Longrightarrow = \overset{par}{\Longrightarrow}; \overset{comm}{\Longrightarrow}; \overset{pc}{\Longrightarrow}; \overset{Unbl}{\Longrightarrow}$. En primer lugar se procede con la evolución paralela, representada por $\overset{par}{\Longrightarrow}$, que controla la ejecución en paralelo de distintas ligaduras activas. La cantidad de trabajo en paralelo que se realiza varía entre la *semántica mínima* y la *semántica máxima*. En el caso de la semántica mínima, no se realiza ningún trabajo especulativo y tan sólo evolucionan las ligaduras que son demandadas a partir de la variable principal *main*. Por contra, si se utiliza la semántica máxima, en cada paso evolucionan en paralelo todas las ligaduras activas del sistema, es decir, se realiza el máximo trabajo especulativo. Tras la evolución en paralelo se ejecuta la regla $\overset{comm}{\Longrightarrow}$ realizando todas las comunicaciones posibles. Análogamente, la regla $\overset{pc}{\Longrightarrow}$ indica que se realizan todas las creaciones de proceso posibles en ese estado. Una vez realizadas todas las transiciones locales posibles, las comunicaciones y las creaciones de proceso, hay que reorganizar las etiquetas de las ligaduras del sistema. Esto se consigue aplicando la regla $\overset{Unbl}{\Longrightarrow}$. Esta reorganización se realiza siguiendo varios pasos: se desbloquean las ligaduras dependientes de una variable que ya ha obtenido un valor, se desactivan las ligaduras que están asociadas a un valor en *whnf*, se bloquean las creaciones de proceso que no han podido realizarse y, por último, se demanda la evaluación de las ligaduras necesarias para realizar las creaciones de proceso y las comunicaciones pendientes.

2.5.2 Semántica Denotacional

Aunque la semántica denotacional de `Jauja` no llega a utilizarse en esta tesis, sí queremos destacar que se trata de una semántica de continuaciones que permite expresar la pereza y los posibles efectos laterales producidos al evaluar una expresión. Es decir, esta semántica no solo se centra en el valor denotacional de una expresión, sino que también refleja explícitamente el paralelismo del lenguaje. Por ejemplo, la denotación de $x_1 \# x_2$ no será únicamente el valor de la aplicación funcional, sino que también reflejará, como efectos laterales, la creación de un proceso y las comunicaciones que se hayan podido realizar. La formalización de la semántica de continuaciones de `Jauja` requiere de la definición de distintos dominios semánticos, y la función de evaluación tiene como tipo:

$$\varepsilon :: \texttt{Exp} \to \texttt{IdProc} \to \texttt{ECont} \to \texttt{Cont},$$

donde hay que indicar la exprexión a evaluar, `Exp`, el proceso en el que se llevará a cabo la evaluación, `IdProc`, y la continuación de expresión que contiene la información de qué hay

que hacer con el valor obtenido, `ECont`. La función de evaluación devolverá una continuación, `Cont`, que acumula los efectos de evaluar la expresión y los de la continuación de expresión.

2.6 Representaciones del λ-cálculo

Tal y como explica Pitts en [Pit13], al definir un lenguaje de programación se especifica una sintaxis muy concreta que servirá para generar los términos (cadenas de símbolos) correctos del lenguaje. Pero muchos detalles de esta sintaxis son irrelevantes para el significado de los programas.

Esta sección se centra en el problema de la α-conversión generado por la sintaxis del λ-cálculo. Uno de los problemas principales que surgen es la captura de variables libres a la hora de realizar una sustitución. Por ello, siempre se habla de términos α-equivalentes, que son aquellos que sólo difieren en el nombre de las variables ligadas. Al realizar una demostración formal, en el caso de que los nombres elegidos generen problemas (captura de nombres), se puede cambiar el término por otro α-equivalente, de modo que las variables ligadas del nuevo término no causen problemas con las variables libres que aparecen en el resto de la demostración. Esta forma de proceder es lo que se conoce como la convención de variables de Barendregt [Bar84].

Sin embargo, y aunque durante muchos años se ha utilizado sin mucha cautela, los nombres elegidos no son tan arbitrarios como se pretendía y, por tanto, la convención de Barendregt no siempre es aplicable, tal y como se explica en [UBN07]. Esto ocurre con cierta frecuencia en pasos de demostraciones por inducción, donde el paso en cuestión puede probarse para variables suficientemente frescas, pero no para una variable arbitraria cualquiera.

A continuación, se exponen distintas alternativas al uso de la notación con nombres.

2.6.1 Notación de de Bruijn

Para dar una formalización del λ-cálculo compatible con las computadoras, de Bruijn propone en [dB72] una notación que denomina libre de nombres (*namefree*), en la que los nombres de las variables son sustituidos por números. Aunque el objeto de estudio de esta notación no fue solventar el problema explicado al comienzo de la sección, cierto es que esta notación evita dichos problemas. Para explicar las ideas del artículo de de Bruijn, vamos a considerar un λ-cálculo formado por variables, abstracciones y aplicaciones, sin declaraciones locales ni constantes, es decir, $t ::= x \mid \lambda x.t \mid a(t,t)$. La idea principal en la que se basa es que los términos α-equivalentes son iguales. El objetivo es lograr una representación única para todos los términos α-equivalentes entre sí. Se presenta a continuación un ejemplo para aclarar los pasos que se siguen.

Ejemplo 1 *Sea la expresión dada por*

$$\lambda x.\lambda y.a(\lambda z.a(a(w,z),t),y)$$

*Para transcribir este término a la notación libre de nombres, se necesita una lista que contenga a las variables libres de la expresión, en este caso w y t. Por ejemplo, podemos elegir $[w,t]$. Se considera entonces el árbol sintáctico de la expresión y se completa en la parte superior con los nodos λw y λt. A cada variable se le asocia un número, la profundidad de referencia (*reference depth*), que indica el número de λ's que hay que pasar al recorrer el árbol hasta llegar a la λ que lleve su nombre. En la Figura 2.12 se*

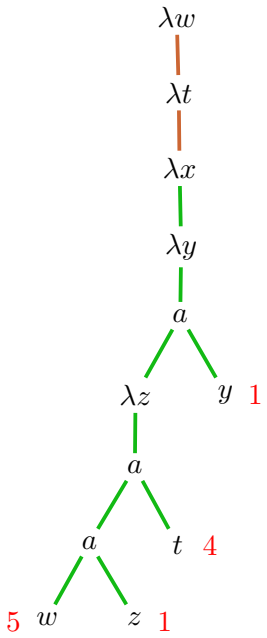


Figura 2.12: Ejemplo de de Bruijn

muestra la construcción del árbol y la profundidad de referencia de cada variable. Además se han marcado con distinto color (marrón) los nodos referentes a las variables libres.

Finalmente se sustituyen los nombres de las variables por los números obtenidos. De este modo la expresión dada con notación libre de nombres será $\lambda.\lambda.a(\lambda.a(a(5,1),4),1)$. □

Pero esta notación libre de nombres tiene una gran desventaja, tal y cómo indica el propio de Bruijn. Pese a su gran utilidad para trabajar en computadoras, resulta poco intuitiva y nada sencilla de usar para el ser humano. Por ejemplo, cada vez que se ejecuta una aplicación, desaparece una λ del árbol sintáctico y hay que recalcular los índices de las variables. Desde el punto de vista de la máquina, esto no es complicado, pues se trata de aplicar ciertas reglas para el ajuste de índices. Sin embargo, si se desea trabajar de forma abstracta sin términos concretos, estos cambios complican considerablemente la sintaxis de la expresión.

2.6.2 Representación localmente sin nombres

Para resolver los problemas derivados de la α-conversión, en esta tesis hemos optado por la representación localmente sin nombres (*locally nameless representation*). Esta notación fue también introducida por de Bruijn [dB72] como alternativa a la notación expuesta en la Sección 2.6.1. Consiste en utilizar índices para las variables ligadas y mantener los nombres de las variables libres. Aunque esta notación se ha utilizado en otros estudios [Gor94, Ler07, ACP+08], destaca el trabajo de Charguéraud [Cha11], que desarrolla una descripción completa de esta representación. En dicho trabajo se muestra la sintaxis del λ-cálculo utilizando esta notación, tal y como se muestra en la Figura 2.13, así como una serie de operaciones necesarias para trabajar con estos términos.

$$t := \texttt{bvar}\ i \mid \texttt{fvar}\ x \mid \texttt{abs}\ t \mid \texttt{app}\ t\ t$$

Figura 2.13: λ-cálculo, representación localmente sin nombres

Entre las principales operaciones sobre los términos representados localmente sin nombres destacan la *apertura* y el *cierre*. La primera sirve para estudiar el cuerpo de una abstracción `abs` t. Al realizar la operación de apertura, t^x, con una variable fresca, el término t se modifica y las variables ligadas (`bvar` i) a la abstracción (`abs` t) de la que provenía el término se convierten en variables libres (`fvar` x). Lo veremos en el siguiente ejemplo:

Ejemplo 2 *Sea el término dado por* $t \equiv$ `abs` u*, donde*

$$u \equiv (\texttt{app}\ (\texttt{abs}\ (\texttt{app}\ (\texttt{bvar}\ 1)\ (\texttt{bvar}\ 0)))\ (\texttt{bvar}\ 0)).$$

En el cuerpo de la abstracción, u*, se observan dos variables que hacen referencia a dicha abstracción. Al abrir dicho cuerpo con la variable* x *se obtiene:*

$$u^x \equiv \texttt{app}\ (\texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x)\ (\texttt{bvar}\ 0)))\ (\texttt{fvar}\ x).$$

□

La operación de cierre es la inversa de la de apertura bajo ciertas condiciones de frescura. Si se quiere construir una abstracción conocido su cuerpo, todas las variables x tendrán que convertirse en variables ligadas.

Ejemplo 3 *Sea el término dado por*

$$u \equiv \texttt{app}\ (\texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x)\ (\texttt{bvar}\ 0)))\ (\texttt{fvar}\ x).$$

Si se quiere construir una abstracción en la que se liguen las variables x*, se tiene*

$$\texttt{abs}\ ({}^{\backslash x}u) \equiv \texttt{abs}\ (\texttt{app}\ (\texttt{abs}\ (\texttt{app}\ (\texttt{bvar}\ 1)\ (\texttt{bvar}\ 0)))\ (\texttt{bvar}\ 0)).$$

□

El problema de esta notación es que se pueden construir términos que no se corresponden con ningún término del λ-cálculo (en notación usual). Para identificar los términos bien formados se define el predicado *localmente cerrado*. Así mismo, en [Cha11] se detallan las funciones de sustitución y variables libres de un término.

En algunas de las reglas que definen los predicados y funciones anteriormente mencionados, Charguéraud utiliza cuantificación cofinita. La utilización de la cuantificación cofinita en reglas ya había sido estudiada por Charguéraud junto con otros autores en [ACP+08]. Se puede decir que la cuantificación cofinita se encuentra entre la cuantificación existencial y la cuantificación universal. En algunas ocasiones, al realizar demostraciones por inducción, es necesario hacer un renombramiento de la variable utilizada para abrir una abstracción, evitando así choques de nombres. Pero la cuantificación cofinita evita estos problemas de choques de nombres, ya que las reglas establecen que la hipótesis se verifica para cualquier variable, salvo una cantidad finita de ellas. En esta tesis se ha utilizado la cuantificación cofinita para expresar algunas de las reglas semánticas en su versión localmente sin nombres.

2.7 Asistentes de demostración

Durante los últimos años se han desarrollado distintas herramientas que permiten trabajar con demostraciones matemáticas. Geuvers resume en [Geu09] la historia e ideas de los asistentes de demostración (*proof assistants*). Hay que diferenciar entre estos y los llamados demostradores automáticos de teoremas (*automated theorem provers*). Mientras que los segundos son sistemas dotados de una serie de procedimientos que permiten demostrar ciertas fórmulas automáticamente, los primeros automatizan los aspectos principales en la construcción de demostraciones pero no son autónomos y necesitan "ser guiados" por un humano en los pasos más controvertidos de la demostración. El usuario utilizará diferentes *tácticas* que guiarán a la máquina para construir la demostración. Aunque los demostradores automáticos han evolucionado mucho y ya son bastante útiles en la práctica, para demostraciones demasiado complejas aún son insuficientes.

Actualmente hay una gran variedad de asistentes de demostración con características ligeramente distintas entre ellos. Entre los más conocidos están Isabelle [isa14], Agda [agd14], PVS [pvs14] y Coq [coq14]. La siguiente tabla resume algunas de las principales características de cada uno de ellos [Wie06]:

Nombre	**Lógica orden superior**	**Tipos dependient.**	**Núcleo pequeño**	**Pruebas automát.**	**Pruebas por reflexión**	**Generac. de código**
Isabelle	Sí	No	Sí	Sí	Sí	Sí
Agda	Sí	Sí	Sí	No	Sí	Sí
PVS	Sí	Sí	No	Sí	No	Sí
Coq	Sí	Sí	Sí	Sí	Sí	Sí

La importancia de la existencia de un núcleo pequeño radica en que sólo hay que verificar que las reglas que lo componen son correctas, ya que el resto de reglas se definen a partir de las que forman el núcleo.

Durante el desarrollo de esta tesis se ha utilizado el asistente Coq para extender algunas de las definiciones y resultados previamente implementados por Charguéraud en [Cha11], referentes a la notación localmente sin nombres, detallada en la Sección 2.6.2.

Capítulo 3

¿Qué hemos obtenido?

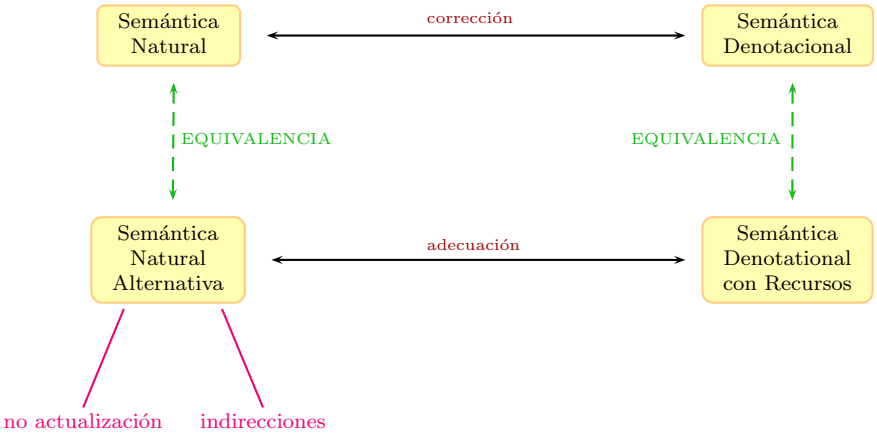
En este capítulo se recopilan y discuten las principales contribuciones de las publicaciones que constituyen esta tesis. Estas se expondrán teniendo en cuenta más la temática a la que corresponden que en el orden cronológico en el que se desarrollaron.

Teniendo en cuenta lo indicado en la Presentación (Capítulo 1), los resultados obtenidos en esta tesis se pueden clasificar en dos secciones: por un lado, el trabajo realizado para demostrar la adecuación computacional comenzada por Launchbury (Sección 2.4); y, por otro, la extensión de algunos resultados a un modelo distribuido.

3.1 Adecuación computacional

En esta sección se explican brevemente los problemas encontrados con respecto a la demostración de la adecuación computacional del trabajo de Launchbury [Lau93]. Posteriormente, nos centraremos en cómo hemos solventado parte de esos problemas; además, en el capítulo de trabajo futuro se explicará cómo estamos trabajando en la solución de los restantes.

En el siguiente esquema aparecen indicadas las semánticas definidas por Launchbury, que han sido presentadas en la Sección 2.4, y la relación entre ellas :



Launchbury centró la demostración de la equivalencia entre su semántica natural y una semántica denotacional estándar en probar la corrección y la adecuación computacional (ver Sección 2.4.1). Como ya se explicó en la Sección 2.2.1, la corrección se basa en ver que el significado de los términos se conserva a lo largo del cómputo, mientras que la adecuación tiene que determinar cuándo existe una reducción; es decir, demuestra que

una expresión es reducible a un valor en la semántica operacional si y sólo si el valor denotacional de dicha expresión está definido. Para probar la adecuación computacional de la semántica natural respecto a la denotacional, Launchbury introdujo dos nuevas semánticas: una semántica natural alternativa y una semántica denotacional con recursos. Tal y cómo explicamos en la Sección 2.4, la primera es una semántica natural en la que no hay actualización de ligaduras y la aplicación se realiza a través de indirecciones, en vez de mediante una β-reducción. La segunda es una semántica denotacional en la que si no hay suficientes recursos los términos no pueden evaluarse. Launchbury demostró la adecuación entre las dos nuevas versiones, sin embargo, sólo comentó brevemente cómo debía hacerse la equivalencia entre las dos semánticas naturales y entre las dos semánticas denotacionales. A la postre las indicaciones dadas para la obtención de estos resultados han resultado ser insuficientes para demostrar dichas equivalencias.

En las próximas dos secciones veremos cómo resolver estas cuestiones. Primero explicamos cómo hemos demostrado la equivalencia entre las semánticas denotacionales. Posteriormente describimos cómo hemos procedido con la parte operacional.

3.1.1 Espacio de funciones con recursos (Publicación P1)

En la semántica denotacional con recursos definida por Launchbury los valores pueden no estar definidos por dos motivos: bien porque sean $\bot$, bien porque no haya recursos suficientes para proceder a la evaluación. Launchbury afirmó que, cuando se dispone de infinitos recursos, esta semántica denotacional y la semántica denotacional estándar producen los mismos valores. Sin embargo, los dominios de definición son diferentes y, por tanto, no se trata en realidad de una igualdad, por lo que hay que buscar una forma de relacionar los valores calculados por cada una de ellas.

En lugar de utilizar el espacio de funciones usual, $D = [D \to D]_\bot$ visto en la Sección 2.3, consideramos la ecuación de dominios $E = [[C \to E] \to [C \to E]]_\bot$, donde C representa los recursos, sobre la que están definidos los valores de la semántica denotacional con recursos. De esta forma, se acota la profundidad de aplicación a la que se puede evaluar. Para construir E hemos seguido los pasos de Abramsky para la construcción de D (Sección 2.3.2), considerando C la solución inicial de la ecuación $C = C_\bot$. Los elementos de C se representan como $\bot$, $S(\bot)$, $S^2(\bot)$,... donde S es la función sucesor. Las *aproximaciones finitas de* E vienen definidas por:

$$\begin{array}{lcl} E_0 & \stackrel{\text{def}}{=} & \{\bot_{E_0}\}, \text{ y} \\ E_{n+1} & \stackrel{\text{def}}{=} & [[C \to E_n] \to [C \to E_n]]_\bot. \end{array}$$

En cada nivel se dispone de más capacidad de definición y los niveles inferiores están contenidos en los niveles superiores:

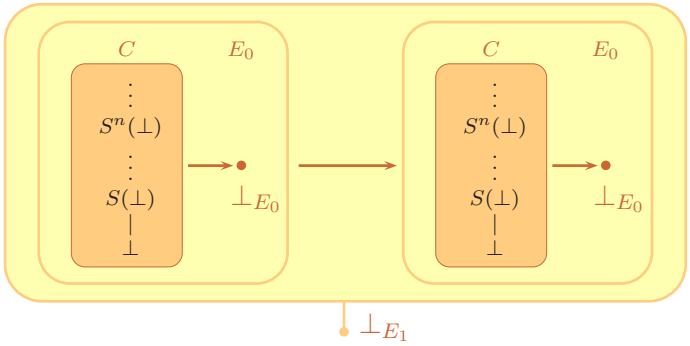
$$\begin{array}{l} E = [[C \to E] \to [C \to E]]_\bot \\ \quad\vdots \\ E_{n+1} = [[C \to E_n] \to [C \to E_n]]_\bot \\ \quad\vdots \\ E_1 = [[C \to E_0] \to [C \to E_0]]_\bot \\ E_0 = \{\bot_{E_0}\} \end{array}$$

A continuación, se muestran gráficamente los primeros niveles de esta construcción. La construcción de E_0 es muy sencilla, pues sólo consta del elemento indefinido:

$$E_0 = \{\bot_{E_0}\}$$

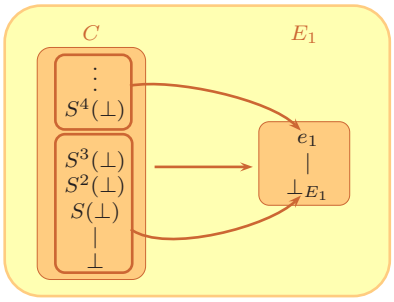
La construcción de E_1 es la siguiente:

$$E_1 = [[C \to E_0] \to [C \to E_0]]_\bot$$



Por un lado se tiene el elemento indefinido de E_1, y por otro las funciones que van de $C \to E_0$ en $C \to E_0$. Sea $A_0 = C \to E_0$, en A_0 hay una única función, a_0, que devuelve el valor indefinido de E_0 sin importar la cantidad de recursos de que se disponga. Por tanto, E_1 consta de dos elementos: el valor indefinido $\bot_{E_1}$ y la función $e_1 : a_0 \mapsto a_0$.

Para entender la construcción de $E_2 = [[C \to E_1] \to [C \to E_1]]_\bot$, consideramos $A_1 = [C \to E_1]$. Este conjunto consta de infinitas funciones en las que, si no hay recursos suficientes, el valor que se devuelve es el indefinido de E_1; mientras que si se dispone de una cantidad adecuada de recursos, devolverá e_1. Por ejemplo, la función $a_{1,4}$ devolverá el valor indefinido si no hay al menos cuatro recursos, mientras que devolverá el valor e_1 en cualquier otro caso, tal y como se muestra a continuación:



Al conjunto formado por todas estas funciones hay que añadirle la función $a_{1,\infty}$, que devuelve siempre el valor indefinido de E_1, independientemente de la cantidad de recursos con que cuente.

De esta forma los elementos de E_2 serán el valor indefinido de E_2, $\bot_{E_2}$, junto con las funciones continuas de $A_1 \to A_1$ que verifican que si $a_{1,m}$ es más definida que $a_{1,n}$, entonces la imagen de $a_{1,m}$ también estará más definida que la de $a_{1,n}$.

Una vez construido el dominio $E = [[C \to E] \to [C \to E]]_\bot$, el siguiente paso es relacionar sus funciones con las de $D = [D \to D]_\bot$, al aplicar una cantidad infinita de recursos. Para ello tomamos la idea de *bisimulación aplicativa* definida por Abramsky (Sección 2.3.3), donde dos funciones se consideran "similares" si producen "valores similares"

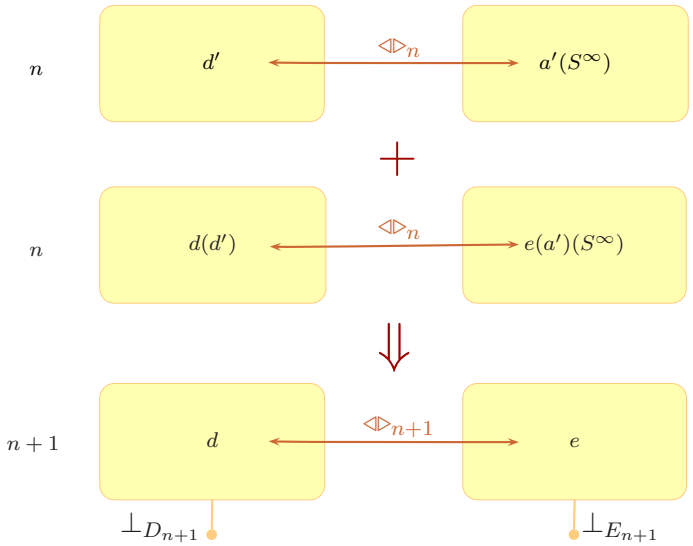


Figura 3.1: Idea de similaridad

al aplicarlas a "argumentos similares", es decir, si tienen un comportamiento "idéntico" dentro de sus dominios de definición. Primero definimos la *similaridad* de funciones por niveles. Por definición, los valores indefinidos del nivel $n+1$-ésimo son *similares*, y dos funciones serán *similares* si para argumentos *similares* en el nivel n-ésimo, producen valores *similares* en el nivel n-ésimo. La Figura 3.1 muestra esta idea.

La relación final $\lhd\rhd$ entre los dominios D y E, se define como la menor relación que verifica que dos valores en D y E están relacionados si sus proyecciones están relacionadas en cada nivel.

Se define una caracterización alternativa de esta relación que expresa que dos valores en D y E están relacionados si, o bien son ambos indefinidos, o si al aplicarlos a argumentos similares se obtienen valores similares, tal y como se indica en la siguiente proposición:

Proposición 1
Siendo $d \in D$, $e \in E$, tenemos que $d \lhd\rhd e$ si y sólo si:

- $(d = \perp_D \wedge e = \perp_E)$, *o bien*
- $(d \neq \perp_D \wedge e \neq \perp_E) \wedge \forall d' \in D. \forall a' \in [C \to E]. d' \lhd\rhd a'(S^\infty) \Rightarrow d(d') \lhd\rhd e(a')(S^\infty)$.

Finalmente hemos aplicado este resultado para demostrar la equivalencia entre las dos semánticas denotacionales propuestas por Launchbury. Para ello hemos extendido el concepto de *similaridad* a entornos, considerando que un entorno ρ de la semántica denotacional estándar es *similar* a un entorno σ de la semántica denotacional de recursos, cuando se dispone de infinitos recursos, si los valores asociados a cada variable en los respectivos entornos son similares. Con esta extensión a los entornos se puede demostrar la equivalencia entre las semánticas denotacionales, tal y como indica el siguiente teorema:

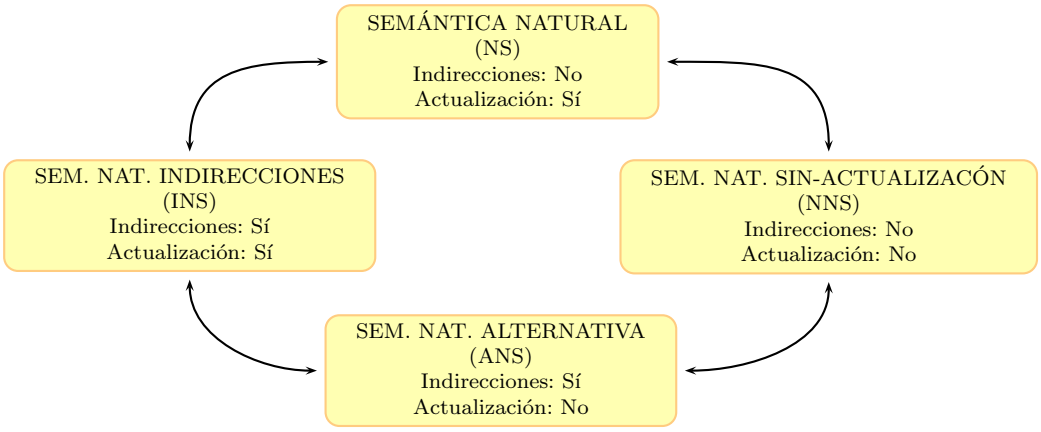
Teorema 3 (Equivalencia de las semánticas denotacionales.)
Si $e \in Exp$ y $\rho \lhd\rhd \sigma$, entonces $[\![e]\!]_\rho \lhd\rhd \mathcal{N}[\![e]\!]_\sigma\ (S^\infty)$.

Resumen de Resultados.

1. Construcción de la solución inicial de la ecuación $E = [[C \to E] \to [C \to E]]_\perp$, donde C representa el dominio de los recursos.
2. Definición de una relación de *similaridad* entre los valores del dominio construido y los valores del espacio de funciones estándar $D = [D \to D]_\perp$.
3. Aplicación del resultado anterior para demostrar la equivalencia entre la semántica denotacional estándar y la semántica con recursos para el λ-cálculo perezoso.

3.1.2 Semántica natural alternativa

Los cambios introducidos por Launchbury en la semántica alternativa (Sección 2.4) resultan tener muchas más consecuencias de las que se pueda pensar en un primer momento. Por un lado, los *heaps* finales tienen un mayor tamaño cuando se evalúa una expresión con la semántica alternativa que al evaluarlos con la semántica original; por otro lado, las expresiones a las que están ligadas las variables no aparecen evaluadas. Por estas razones hemos tratado por separado las modificaciones que se introducen en la semántica alternativa, dando lugar a dos semánticas intermedias, tal y como se indica en la siguiente figura:



La *semántica natural con indirecciones* (INS) mantiene la actualización de ligaduras pero introduce indirecciones a la hora de evaluar las aplicaciones. La *semántica natural sin actualización* (NNS), tal y como indica su nombre, no actualiza las ligaduras del *heap* y no introduce indirecciones. Tratando cada cambio por separado es más sencillo centrarse en las diferencias que se producen en los *heaps* finales y buscar un modo de relacionar las semánticas.

Los cambios introducidos por las reglas semánticas alternativas no son las únicas dificultades encontradas a la hora de establecer la equivalencia entre las semánticas. Existe un problema añadido derivado de la notación con nombres. La α-conversión es una complicación implícita cuando se trabaja con semánticas operacionales. Para evitar las dificultades de una notación con nombres, hemos realizado el estudio entre semánticas con una representación *locally nameless* (localmente sin nombres [Cha11]), donde los nombres de las variables ligadas se sustituyen por índices de de Bruijn [dB72] pero se conservan los nombres de las variables libres, tal y cómo se explica en la Sección 2.6.2. Esta representación facilita además la formalización en asistentes de demostración.

$$
\begin{array}{lcl}
x \in Id & & i, j \in \mathbb{N} \\
v \in Var & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in LNExp & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

Figura 3.2: Sintaxis localmente sin nombres

Representación localmente sin nombres (Publicación P2 y TechRep TR1)

Se procede a explicar cómo hemos extendido con declaraciones locales recursivas la representación localmente sin nombres del λ-cálculo dada por Charguéraud en [Cha11]. Al eliminar los nombres de las variables ligadas se evita tener que trabajar con clases de equivalencia y elegir un representante de ellas, pues todos los términos cerrados que sean semánticamente iguales se representan de manera única.

Como se ha visto en la Sección 2.6.2, Charguéraud explica y desarrolla las ventajas de la representación localmente sin nombres para un lenguaje con variables, abstracciones y aplicaciones. Como se observa en la Figura 2.4 de la Sección 2.4, en el cálculo utilizado por Launchbury, además de las expresiones anteriores se consideran declaraciones locales recursivas. Tal y como indica Charguéraud, las variables ligadas en estos casos se representan con dos índices: el primero indica a qué constructor está ligada —una abstracción o una declaración local— tal y cómo se explica en la Sección 2.6.1 y puede verse en el Ejemplo 1; el segundo índice indica, dentro del constructor que liga la variable, a qué expresión nos referimos (cuando el constructor es una declaración local, indica a qué variable local se hace la referencia). El resultado es la sintaxis que se muestra en la Figura 3.2, donde *Var* representa un conjunto de *variables* que pueden ser ligadas (`bvar` i j) o libres (`fvar` x).

El siguiente ejemplo ilustra cómo se determinan los índices de las variables ligadas.

Ejemplo 4 *La expresión e es una λ-abstracción cuya expresión interna es un* let *con dos definiciones locales.*

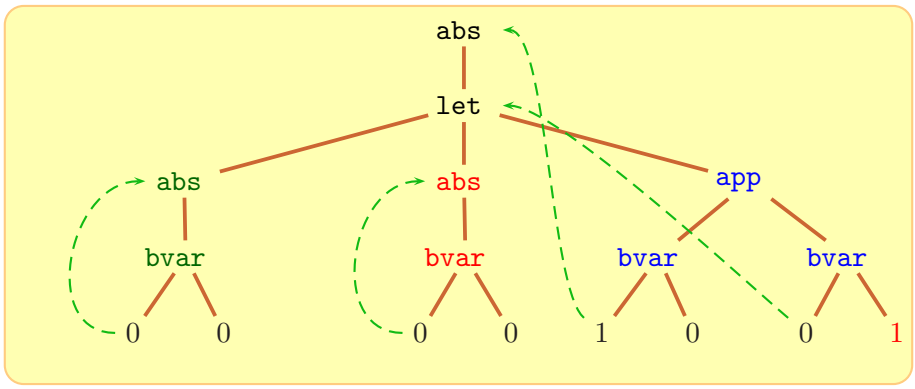
$$
\begin{array}{rl}
e \equiv \lambda z.\texttt{let} & x_1 = \lambda y_1.y_1, \\
& x_2 = \lambda y_2.y_2, \\
\texttt{in} & (z\ x_2)
\end{array}
$$

La representación de esta expresión con la notación localmente sin nombres viene dada por el término t:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

En verde aparece la expresión ligada a x_1, en rojo la de x_2 y en azul el término principal.

Si nos fijamos en el árbol sintáctico veremos cómo los primeros índices señalan a los constructores que ligan:



Nótese que el segundo índice de la variable que señala al let *es de color rojo para indicar que, dentro de las declaraciones locales de este constructor, está haciendo referencia a la que aparece con dicho color.* □

Aunque Charguéraud indica cómo extender el lenguaje con las declaraciones locales, el estudio de esta extensión no lo desarrolla plenamente: se explica cómo realizar la apertura de términos y se dan las reglas para la clausura local, pero el resto de operadores y funciones sobre términos no están considerados en [Cha11]. Por ese motivo, hemos extendido varias definiciones como la clausura de términos, el predicado de clausura local en un nivel k, las variables libres de un término, la sustitución y el concepto de frescura de nombres, entre otras. También hemos extendido varios resultados como el que indica que, bajo ciertas condiciones, la apertura y cierre de un término son funciones inversas.

Una vez establecidos todos los operadores y funciones necesarios para trabajar con la notación localmente sin nombres, hemos reformulado las reglas semánticas del modelo de Launchbury (Figura 2.4 en la Sección 2.4) utilizando dicha notación. En primer lugar se han redefinido los *heaps* como conjuntos de ligaduras o pares de la forma $(\texttt{fvar}\ x, t)$ y todos los conceptos relacionados con los *heaps* también se han extendido a esta notación: dominio de un *heap*, nombres que aparecen en un *heap*, *heap* bien formado y sustituciones en *heaps*. Además hemos demostrado algunas propiedades derivadas de esta notación.

Las reglas de la semántica natural son reglas inductivas en las que intervienen *heaps* y λ-términos. Puesto que la notación localmente sin nombres permite formar términos incorrectos (en el sentido de que no corresponden a ningún término del λ-cálculo) hemos añadido algunas premisas para asegurar que las reglas inductivas se restringen a *heaps* bien formados y a términos *localmente cerrados* (concepto explicado en la Sección 2.6.2).

De la nueva versión de las reglas semánticas sólo merece destacar aquí la regla LN-Let (Figura 3.3) para las declaraciones locales. Las tuplas se expresan como $\overline{t}$ y $\overline{x}$ según se refieran a términos o variables. Para evaluar una expresión $\texttt{let}\ \overline{t}\ \texttt{in}\ t$ se añaden al *heap* las declaraciones locales $\overline{t}$, para evaluar t en el contexto ampliado. En la notación localmente sin nombres, y puesto que no están reflejados en el término los nombres de las declaraciones locales, es necesario elegir una lista de nombres $\overline{x}$ que sean frescos. Con estos nombres se abren los términos locales y el cuerpo. La evaluación de $t^{\overline{x}}$ produce un *heap* y un valor finales, siendo ambos dependientes de los nombres elegidos. Por tanto, en el *heap* final destacamos los nombres introducidos al evaluar el *let*, $\overline{x}$, y el resto de nombres que se denotan por $\overline{z}$. En la regla LNLet se utiliza la cuantificación cofinita que, tal y como se explica en [Cha11], tiene la ventaja de no tener que especificar explícitamente las condiciones laterales de frescura de $\overline{x}$. Dichas condiciones están implícitas en el conjunto finito L. En este conjunto se recogen los nombres que deben evitarse durante la reducción, en particular por la razón de que ya aparecen en otra parte de la derivación. Sin embargo,

$$\text{LNLET}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \quad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Figura 3.3: Declaración local de variables (notación localmente sin nombres)

nuestra regla cofinita es diferente a las que se describen en [Cha11], pues los nombres elegidos aparecen también en la conclusión. Es decir los nombres elegidos, $\overline{x}$, pueden ser reemplazados por cualquier lista de nombres $\overline{y}$ que no estén en L.

Después de traducir a la notación localmente sin nombres las reglas semánticas correspondientes a la semántica natural y a su versión alternativa[1], hemos demostrado diversas propiedades, entre las que destacaremos las siguientes:

Regularidad: asegura que los juicios producidos por estos sistemas de reducción sólo producen *heaps* bien formados y términos localmente cerrados;

Renombramiento: indica que la evaluación de un término es independiente de los nombres frescos elegidos durante la reducción;

Introducción: establece, para cada regla cofinita, que se tiene una versión existencial que también es correcta.

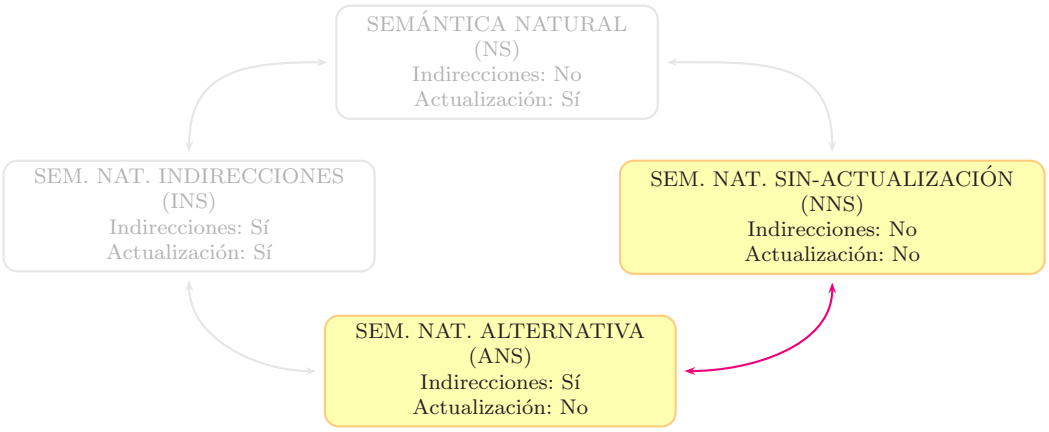
Resumen de Resultados.

1. Representación localmente sin nombres del λ-cálculo extendido con declaraciones locales recursivas.
2. Versión localmente sin nombres de las reglas de la semántica natural de Launchbury y su versión alternativa.
3. Propiedades de los sistemas de reducción (lemas de regularidad, introducción y renombramiento).

Relación de indirecciones (Publicación P3 y TechRep TR2)

Una vez establecidas las reglas semánticas en la nueva notación podemos investigar la equivalencia entre la semántica sin actualización (NNS) y la semántica alternativa (ANS), es decir, demostrar que al evaluar un término en un *heap* determinado el resultado obtenido con una y otra semántica es el "mismo". Esta equivalencia es la que aparece resaltada en color en el siguiente diagrama:

[1] En realidad, la traducción a la notación localmente sin nombres de las reglas alternativas no aparece en P2 sino en P3, que se comenta en el siguiente apartado. Pero hemos considerado más conveniente hacer la referencia en esta sección dedicada a la representación localmente sin nombres.



Las semánticas NNS y ANS utilizan las mismas reglas de derivación excepto por la regla de aplicación, que en el caso de NNS aplica la regla LNApp, donde la aplicación se realiza mediante una β-reducción, y en el caso de ANS aplica la regla ALNApp, en la que se introduce una *indirección* en el *heap*. Aunque esta diferencia parece inocua, lo primero que se observa es que los *heaps* finales obtenidos tras aplicar las reglas de ANS pueden ser más grandes que los obtenidos tras aplicar las reglas de NNS, ya que cada vez que se evalúa una aplicación, se introduce una ligadura "extra" en el *heap*. Esta ligadura es una indirección, es decir, una variable ligada a otra variable. Por ello los *heaps* finales obtenidos no pueden ser exactamente los mismos y hay que estudiar qué relación existe entre ellos. Teniendo en cuenta este dato, lo primero en lo que se pensó fue en eliminar las indirecciones y comprobar si los *heaps* finales obtenidos eran iguales. Sin embargo, esto no es suficiente, ya que en el *heap* final puede haber términos que dependan de las indirecciones que se han eliminado.

En el siguiente ejemplo se muestra este problema, donde $\Downarrow^N$ representa la evaluación con NNS y $\Downarrow^A$ la evaluación con ANS:

Ejemplo 5 *Considérense los siguientes términos:*

$$t \equiv \texttt{let abs (bvar 0 0) in app (abs } s\texttt{) (bvar 0 0)}$$
$$s \equiv \texttt{let abs (bvar 0 0), app (bvar 0 0) (bvar 1 0) in abs (bvar 0 0)}$$

Al evaluar t en el contexto de un heap *vacío con las dos semánticas en estudio, se obtiene en ambos casos* `abs (bvar 0 0)` *como valor final y los siguientes* heaps*, respectivamente:*

$\{\,\} : t \Downarrow^N$

$x_0 \mapsto \texttt{abs (bvar 0 0)}$,
$x_1 \mapsto \texttt{abs (bvar 0 0)}$,
$x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } x_0\texttt{)}$

$\{\,\} : t \Downarrow^A$

$x_0 \mapsto \texttt{abs (bvar 0 0)}$,
$x_1 \mapsto \texttt{abs (bvar 0 0)}$,
$x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } y\texttt{)}$
$y \mapsto \texttt{(fvar } x_0\texttt{)}$

Se observa que el heap *obtenido al evaluar el término con la semántica alternativa contiene una ligadura más, y que el término asociado a la variable x_2 depende de dicha ligadura extra.* □

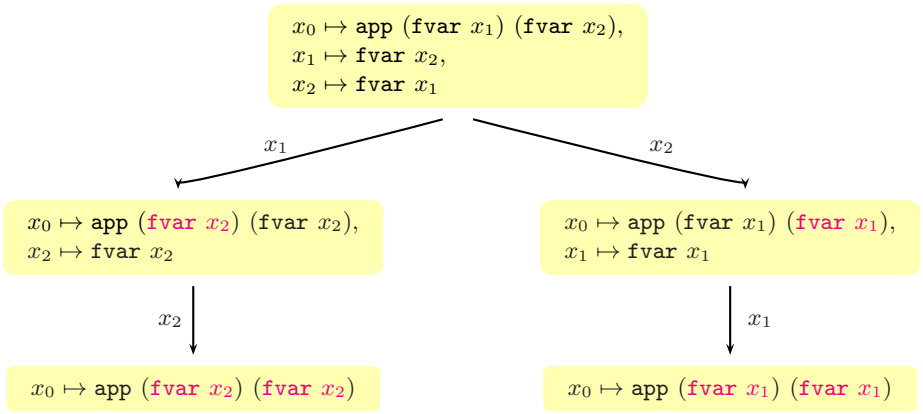
Por tanto, la relación que buscamos entre *heaps* no puede consistir sólo en eliminar las indirecciones, sino que también es necesario realizar una sustitución para cambiar todas

las apariciones que haya en el *heap* con el nombre de la variable que se va a eliminar por el nombre de la variable a la que está ligada.

Se pensó en encontrar un método para detectar, de entre todas las indirecciones, cuál o cuáles provenían de aplicar la regla ALNApp, pero hemos considerado que es más interesante no focalizar el problema en concreto, sino darle una visión más extensa que contenga como caso particular el caso específico en estudio. Para ello hemos buscado una relación entre *heaps* que esté basada en indirecciones, independientemente de su procedencia. La idea es que dos *heaps* estén relacionados si al eliminar indirecciones de uno de ellos se obtiene el *heap* que es más pequeño en tamaño.

La pregunta que surge de forma natural es si influye el orden en el que se eliminen las indirecciones. En el caso de referencias cruzadas, la respuesta es afirmativa, como muestra el siguiente ejemplo:

Ejemplo 6 *El* heap $\Gamma = \{x_0 \mapsto \texttt{app}\ (\texttt{fvar}\ x_1)\ (\texttt{fvar}\ x_2), x_1 \mapsto \texttt{fvar}\ x_2, x_2 \mapsto \texttt{fvar}\ x_1\}$, *contiene dos indirecciones. A continuación se muestra que el orden de eliminación de estas dos ligaduras afecta al resultado final:*



□

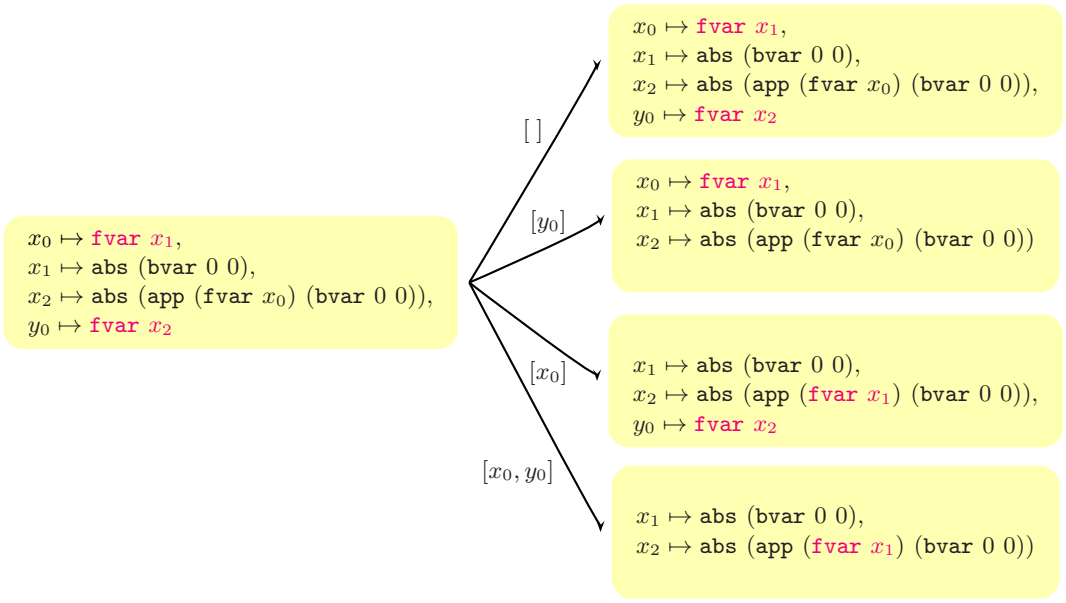
Nótese, sin embargo, que en el ejemplo anterior ambas variables x_1 y x_2 están indefinidas en el *heap* final. Por ello dos *heaps* se considerarán relacionados si son iguales salvo por variables indefinidas.

De esta forma, definimos la *equivalencia de términos en un contexto dado*, que da lugar a la *equivalencia de* heaps *en un contexto dado*. En esta última equivalencia se basa la *relación por indirecciones* ($\gtrsim_I$) entre *heaps*. En concreto, dos *heaps* están relacionados por indirecciones si al eliminar ciertas indirecciones del *heap* mayor se obtiene el *heap* menor, salvo por el nombre de variables indefinidas, es decir, variables que no están en el dominio del *heap* pero pueden aparecer en los términos de las ligaduras (lados derechos).

Así un *heap* puede estar relacionado con varios *heaps*, tal y como muestra el siguiente ejemplo.

Ejemplo 7 *A continuación, se muestran todos los* heaps *que están relacionados por indirecciones con el siguiente:*

$$\Gamma = \{\ x_0 \mapsto \texttt{fvar}\ x_1, x_1 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{bvar}\ 0\ 0)),\ y_0 \mapsto \texttt{fvar}\ x_2\ \}$$



□

Extendemos esta relación a pares (*heap*, término) y enunciamos y demostramos el teorema que establece que si un término evalúa en un contexto dado con NNS, también lo hace con ANS, y viceversa; y además, los pares (*heap*, término) finales están relacionados por indirecciones.

Teorema 4 (Equivalencia ANS y NNS.)

EQ_AN $\quad \Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in LNHeap \,.\, \exists w_N \in LNVal \,.\, \Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \gtrsim_I (\Delta_N : w_N)$

EQ_NA $\quad \Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in LNHeap \,.\, \exists w_A \in LNVal \,.\, \exists \overline{x} \subseteq \mathtt{dom}(\Delta_N) - \mathtt{dom}(\Gamma) \,.\, \exists \overline{y} \subseteq Id \,.\, |\overline{x}| = |\overline{y}| \wedge$
$\Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \gtrsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$

En la segunda parte del teorema, EQ_NA, se indica que es posible que haga falta un renombramiento, pero éste afectará sólo a los nombres que han sido añadidos durante la evaluación. Con NNS, algunos de los nombres del término a evaluar pueden desaparecer durante la evaluación y ser introducidos de nuevo como "frescos"; sin embargo, al evaluar con ANS esto no puede ocurrir, tal y como muestra el siguiente ejemplo:

Ejemplo 8 *Considérese el término*

$$t \equiv \mathtt{let\ abs\ (bvar\ 1\ 1), let\ abs\ (bvar\ 0\ 0)\ in\ (bvar\ 0\ 0)\ in\ app\ (bvar\ 0\ 0)\ (fvar}\ z)$$

Al evaluarlo en un contexto vacío con NNS, la variable z desaparece del término (y del heap*); por el contrario, al evaluarlo con ANS se introduce en el* heap *mediante una indirección. A continuación, se muestra la derivación de la evaluación del término con NNS:*

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\{\}:t}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{fvar}\ z)}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{fvar}\ x_0}
\quad
\dfrac{
\dfrac{
\dfrac{}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1}
\quad
\dfrac{}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{fvar}\ x_1}
\quad
\dfrac{
\dfrac{}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)}
\quad
\vdots
\quad
\left\{\begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array}\right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
}{
\left\{\begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array}\right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)}
$$

Al realizar la β-reducción, el nombre z ya no aparece más, por lo que al introducir la variable fresca x_2 en el heap*, se podría haber elegido como nombre z.*

Sin embargo, al realizar la evaluación con la semántica alternativa, se introduce z en el heap *mediante una indirección, y ya no puede ser elegido cuando se necesite un nombre "fresco".*

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\{\}:t}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{fvar}\ z)}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{fvar}\ x_0}
\quad
\dfrac{
\dfrac{
\dfrac{}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1}
\quad
\dfrac{}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1}
}{\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), y \mapsto \texttt{fvar}\ z\} : \texttt{fvar}\ x_1}
\quad
\dfrac{
\dfrac{}{\left\{\begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), y \mapsto \texttt{fvar}\ z, \\ x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0) \end{array}\right\} : \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)}
\quad
\vdots
\quad
\left\{\begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array}\right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
}{
\left\{\begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array}\right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
}
}{
\left\{\begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array}\right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
}
$$

□

La demostración del teorema no puede realizarse directamente mediante una inducción por reglas, pues en las subderivaciones el *heap* y los términos de los que se parte no son iguales, sino que están relacionados por indirecciones. Por esta razón, es necesaria la demostración de un resultado más general, en el que se establece la relación entre las derivaciones de las dos semánticas en estudio partiendo de pares (*heap*, término) relacionados por indirecciones (Proposición 2). Puesto que en las derivaciones se van introduciendo variables frescas en el *heap*, siempre que haya una derivación habrá de hecho infinitas, que se diferencian en los nombres frescos elegidos. La proposición indica que partiendo

de dos pares (*heap*, término) relacionados por indirecciones, y en el caso de que se puedan obtener derivaciones con ANS para el *heap* más grande, una de estas derivaciones estará relacionada por indirecciones con una derivación con NNS del *heap* más pequeño, y viceversa.

Proposición 2

EQ_IR_AN $(\Gamma_A : t_A) \succsim_I (\Gamma_N : t_N) \wedge \forall \overline{x} \notin L \subseteq Id \,.\, \Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}}$
$\wedge \, {}^{\setminus \overline{x}}(\overline{s}_A{}^{\overline{x}}) = \overline{s}_A \wedge {}^{\setminus \overline{x}}(w_A{}^{\overline{x}}) = w_A$
$\Rightarrow \exists \overline{y} \notin L \,.\, \exists \overline{s}_N \subset LNExp \,.\, \exists w_N \in LNVal \,.$
$\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}} \wedge {}^{\setminus \overline{z}}(\overline{s}_N{}^{\overline{z}}) = \overline{s}_N \wedge {}^{\setminus \overline{z}}(w_N{}^{\overline{z}}) = w_N \wedge \overline{z} \subseteq \overline{y}$
$\wedge \, ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$

EQ_IR_NA $(\Gamma_A : t_A) \succsim_I (\Gamma_N : t_N) \wedge \forall \overline{x} \notin L \subseteq Id \,.\, \Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}}$
$\wedge \, {}^{\setminus \overline{x}}(\overline{s}_N{}^{\overline{x}}) = \overline{s}_N \wedge {}^{\setminus \overline{x}}(w_N{}^{\overline{x}}) = w_N$
$\Rightarrow \exists \overline{z} \notin L \,.\, \exists \overline{s}_A \subset LNExp \,.\, \exists w_A \in LNVal \,.$
$\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}} \wedge {}^{\setminus \overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \wedge {}^{\setminus \overline{y}}(w_A{}^{\overline{y}}) = w_A \wedge \overline{z} \subseteq \overline{y}$
$\wedge \, ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$

La demostración de esta proposición requiere de varios lemas técnicos en los que mostramos cómo se transmite la relación por indirecciones a las subderivaciones, para así poder demostrar los casos inductivos. Por ejemplo, si dos pares (*heap*, término) están relacionados mediante indirecciones y los términos son aplicaciones, también estarán relacionados por indirecciones los *heaps* correspondientes con los cuerpos de sus respectivas aplicaciones. O si los términos son declaraciones locales, también estarán relacionados los *heaps* ampliados con dichas declaraciones y los términos principales.

Resumen de Resultados.

1. Relación de equivalencia entre *heaps* que definen las mismas variables libres pero cuyas clausuras pueden diferir en las variables libres indefinidas.
2. Preorden que relaciona dos *heaps* cuando el primero puede transformarse en el segundo mediante la eliminación de indirecciones ($\succsim_I$).
3. Extensión del preorden para pares (*heap*, término).
4. Equivalencia de ANS y NNS.

3.2 Modelo Distribuido (Publicación P4)

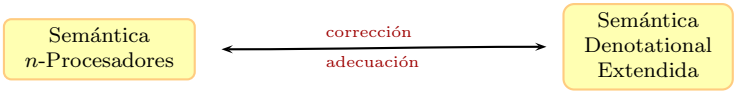
Como se comentó en la presentación (Capítulo 1), uno de los objetivos principales de esta tesis consistía en establecer ciertas propiedades de la semántica operacional de un modelo distribuido en el que se dispone de n procesadores con respecto a una semántica denotacional estándar como, por ejemplo, la corrección y adecuación computacional. Históricamente, este objetivo fue el primero que se abordó y de hecho la búsqueda de las demostraciones necesarias dio lugar a los resultados explicados en las secciones anteriores.

Tomando como base el lenguaje `Jauja`, introducido por Hidalgo-Herrero en [Hid04] y explicado en la Sección 2.5, extendemos la sintaxis dada en [Lau93] con la *aplicación paralela* que dará lugar a la creación de un nuevo proceso. La sintaxis de estas expresiones extendidas (*EExp*) se muestra en la Figura 3.4. Se trata de una sintaxis restringida (a semejanza de lo explicado en la Sección 2.4) en la que tanto las subexpresiones de las aplicaciones, como el cuerpo de las construcciones *let*, son variables. De esta forma se simplifican las definiciones de las reglas semánticas.

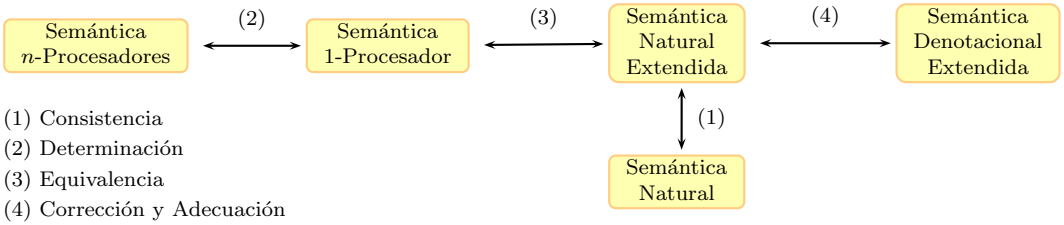
$$
\begin{array}{rcl}
x,\ y & \in & \mathit{Var} \\
E & \in & \mathit{EExp} \\
E & ::= & x \mid \lambda x.E \mid x\ y \mid x\#y \mid \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x
\end{array}
$$

Figura 3.4: Sintaxis

Revisamos las reglas de la semántica operacional definidas en [Hid04] (comentadas en la Sección 2.5) y las restringimos a *EExp* (semántica con n procesadores). Asímismo extendemos la semántica denotacional estándar para dar significado a la nueva expresión de aplicación paralela (semántica denotacional extendida), y demostramos la equivalencia entre ambas, en términos de corrección y adecuación computacional:



La demostración de esta equivalencia no se realiza directamente, sino estudiando las relaciones existentes entre una serie de semánticas intermedias. El siguiente esquema muestra los distintos pasos a seguir y que se irán explicando brevemente a lo largo de esta sección:



Dado que Launchbury ya había establecido en [Lau93] la corrección y adecuación computacional de las semánticas correspondientes para un lenguaje más sencillo, se deseaba aprovechar dicho trabajo y extender sus resultados al caso de la aplicación paralela. Por ello se introdujo una semántica intermedia, *semántica natural extendida* (ENS), que es una extensión de la semántica natural de Lauchbury para trabajar con creaciones de procesos y comunicaciones. Aunque la creación de procesos es impaciente, éstos sólo se crean bajo la condición de que las variables que sean necesarias para evaluar la aplicación correspondiente no estén bloqueadas. El primer problema que nos encontramos es que, debido a la regla variable, algunas ligaduras desaparecen del *heap*, y no se dispone de dicha información. Por ello extendemos los *heaps* para que consten de dos partes: $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$, donde la segunda parte (Γ^B) almacena las ligaduras bloqueadas o, dicho de otro modo, que ya han sido demandadas. Estas ligaduras pueden provenir, o bien de la regla aplicación, o bien de la regla variable. El segundo problema surge del hecho de que se pierde el nombre al que van ligadas las expresiones que están siendo evaluadas. Por ello, la expresión que se evalúa en cada momento aparece ahora ligada a un nombre y, por tanto, los juicios se escriben ahora como

$$\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W.$$

Para indicar que las creaciones de procesos deben realizarse tan pronto como sea posible, en algunas reglas semánticas (la regla para las λ-abstracciones y la regla para las declaraciones

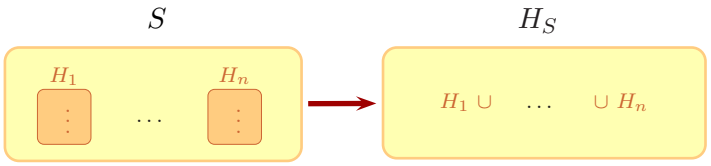


Figura 3.5: Conversión de un modelo distribuido a uno que no lo es

locales) exigimos que los *heaps* estén *saturados*, o lo que es lo mismo, que no tengan creaciones de proceso pendientes.

Demostramos que esta extensión es *consistente* con la semántica natural de partida, es decir, si se consideran expresiones sin aplicación paralela, los valores resultantes deben ser los mismos al utilizar ambos sistemas de derivación, tal y cómo se indica en el siguiente teorema.

Teorema 5 (Consistencia.) *Para todo* $e \in Exp$ *se tiene que* $\Gamma : e \Downarrow \Delta : w$ *si y sólo si* $\langle \Gamma, \Gamma^B \rangle : x \mapsto e \Downarrow \langle \Delta, \Gamma^B \rangle : x \mapsto w$*, donde* x *es una variable completamente fresca en la derivación, y* Γ^B *es disjunto con respecto a* Γ *y* Δ*.*

Puesto que en el caso de ENS solamente se dispone de un procesador, para relacionar esta semántica con la de n procesadores, introducimos otra semántica intermedia de paso corto con creación de procesos y comunicaciones, pero para un único procesador: la *semántica con* 1*-procesador*.

La *semántica con n-procesadores* representa un modelo distribuido en el que un sistema formado por varios procesos evoluciona hasta obtener, al menos, el valor de la variable principal. Recordemos que tenemos dos tipos de reglas: las reglas locales que indican cómo evoluciona cada uno de los procesos representados por *heaps* etiquetados; y las reglas globales que regulan la creación de procesos y la comunicación entre ellos. En el caso de la semántica con 1-procesador, sólo puede haber una ligadura activa en cada momento. Puesto que sólo puede evolucionar una ligadura, en vez de tener dos niveles de reglas, hay uno único en el que mantenemos la creación impaciente de procesos e imponemos un orden en la evaluación de las ligaduras, que es compatible con la semántica mínima descrita en la Sección 2.5.1.

Si queremos comparar la semántica con n-procesadores y la semántica con 1-procesador para comprobar que producen en efecto el mismo valor para la variable principal, tendremos que estudiar cómo pasar de un modelo distribuido a otro que no lo es. Construir un sistema no distribuido a partir de un sistema distribuido es sencillo, pues basta con unir todas las ligaduras de los distintos procesos en un único *heap*. Las ligaduras potencialmente activas en el modelo distribuido pasarán a estar inactivas en el *heap* único, salvo solo una. La única ligadura activa en el *heap* se corresponderá con aquella que es desarrollable en el momento actual, y que viene dada por la función $EB(H)$ (*evolutionary bindings*), que calcula las ligaduras desarrollables de un *heap*. Esta función no es fija, sino que depende de la semántica que se esté utilizando. En este caso, ya se ha mencionado que se trata de la semántica mínima, por lo tanto sólo puede haber una ligadura desarrollable en cada momento: la que viene demandada directamente por la variable principal *main*. La Figura 3.5 muestra esquemáticamente cómo convertir un sistema distribuido de *heaps* en un *heap* único.

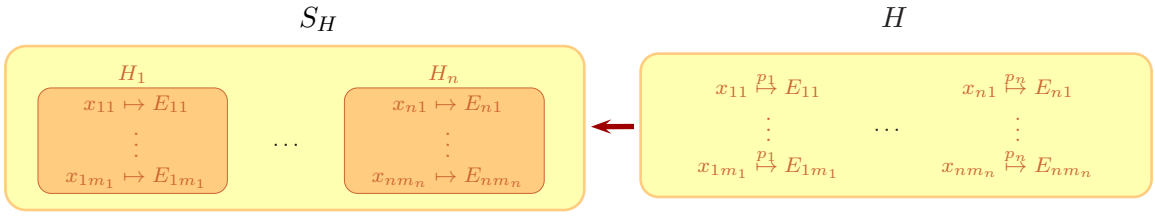


Figura 3.6: Conversión de un modelo no distribuido a uno que sí lo es

También se ha de poder ir en el sentido contrario y construir un sistema distribuido partiendo de un *heap* de ligaduras. Está parte es más delicada, pues es necesario saber a qué proceso pertenece cada ligadura. Para ello incluimos una anotación que indica a qué proceso corresponde cada una de ellas. Esta anotación no interfiere con las reglas semánticas. El esquema de la Figura 3.6 muestra cómo pasar de un *heap* a su sistema asociado.

El teorema de determinación establece la equivalencia entre la semántica con n-procesadores y la de 1-procesador, tomando la semántica mínima para el modelo con n procesadores. Nótese que los sistemas se representan como $S = \langle p_i, H_i \rangle_{i=0}^{n}$, donde cada par $\langle p_i, H_i \rangle$ representa un proceso, siendo p_i su nombre y H_i su *heap*.

Teorema 6 (Determinación.) *Sea $E \in EExp$.*

1. *Si $\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* S \Longrightarrow^* S'$, entonces existe una derivación $\{main \overset{A}{\mapsto} E\} \Longrightarrow_1^* H_S \Longrightarrow_1^* H_{S'}$, donde H_S (respectivamente $H_{S'}$) es el* heap *construido a partir de S (respectivamente S').*

2. *Si $\{main \overset{A}{\mapsto} E\} \Longrightarrow_1^* H \Longrightarrow_1^* H'$, entonces existe un cómputo $\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* S_H \Longrightarrow^* S_{H'}$, donde S_H (respectivamente $S_{H'}$) es el sistema de procesos asociado al* heap *H (respectivamente H').*

A continuación, consideramos la equivalencia entre la semántica con 1-procesador (de paso corto) y la semántica natural extendida (de paso largo). Para establecer dicha equivalencia estudiamos la relación entre los *heaps* etiquetados producidos por la semántica con 1-procesador y los *heaps* extendidos de ENS. No es complicado pasar de unos a otros, pues las ligaduras inactivas de los *heaps* obtenidos con la semántica de paso corto se corresponden con la primera parte del *heap* extendido de ENS, mientras que las ligaduras bloqueadas se corresponden con la segunda parte. Además, la única ligadura activa dará lugar a la expresión que está siendo evaluada. De forma similar, pero a la inversa, se puede construir un *heap* para la semántica con 1-procesador partiendo de un *heap* extendido. La Figura 3.7 muestra cómo pasar de unos *heaps* a otros.

De esta forma, es sencillo establecer el teorema de equivalencia entre ambas semánticas.

Teorema 7 (Equivalencia ENS y semántica con 1-Procesador.) *Sea $E \in EExp$.*

1. *Si $H + \{\theta \overset{A}{\mapsto} E\} \Rightarrow_1^* H' + \{\theta \overset{A}{\mapsto} W\}$, entonces $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, donde $\overline{\Gamma}$ y $\overline{\Delta}$ son los* heaps *extendidos asociados a H y H', respectivamente.*

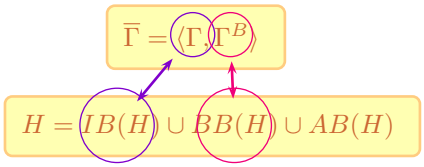


Figura 3.7: Esquema de conversión de *heaps*

2. *Si* $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, *entonces* $H + \{\theta \overset{A}{\mapsto} E\} \Rightarrow_1^* H' + \{\theta \overset{A}{\mapsto} W\}$, *donde* H *y* H' *son los* heaps *extendidos asociados a* $\overline{\Gamma}$ *y* $\overline{\Delta}$, *respectivamente.*

Una vez establecidas las equivalencias entre las distintas semánticas operacionales, procedemos a demostrar la *corrección* y la *adecuación computacional* entre las versiones extendidas de la semántica natural y de la semántica denotacional.

Antes de enunciar los teoremas principales se verá brevemente cómo hemos extendido la semántica denotacional de Abramsky para dar significado a los canales y a la aplicación paralela. La función semántica es ahora

$$[\![-]\!]_\rho : \ \mathit{EExp} \cup \mathcal{C}\mathit{han} \to \mathit{Env} \to \mathit{Value}$$

siendo $\rho \in \mathit{Env} = \mathit{Var} \cup \mathcal{C}\mathit{han} \mapsto \mathit{Value}$ un entorno de identificadores a valores. La denotación de la nueva expresión de aplicación paralela coincide con la denotación de la aplicación, es decir, $[\![x\#y]\!]_\rho = [\![x\ y]\!]_\rho$ ya que el valor final obtenido es el mismo.

El teorema de corrección establece que el significado de una expresión no varía durante su evaluación. Es más, los *heaps* sólo pueden aumentar de tamaño al añadir nuevas ligaduras, o bien sufrir un refinamiento de las ligaduras existentes, es decir, ser actualizados con los valores calculados. Consideremos *Heap*, el dominio de los *heaps* sin etiquetar, es decir, conjuntos de ligaduras no etiquetadas; $\rho \leq \rho'$, el orden sobre los entornos definido por Launchbury y explicado en la Sección 2.4; y la función $\{\!\{-\}\!\} : \ \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$, para extender los entornos a partir de las ligaduras de un *heap*, también explicada en la Sección 2.4.

Teorema 8 (Corrección de la semántica natural extendida.) *Sea* $E \in \mathit{EExp} \cup \mathcal{C}\mathit{han}$, $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$, $\overline{\Delta} = \langle \Delta, \Delta^B \rangle \in \mathit{EHeap}$, *y* $\theta \notin \mathit{dom}(\overline{\Gamma})$.
Si $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, *entonces para cada entorno* ρ, $[\![E]\!]_{\{\!\{\Gamma\}\!\}_\rho} = [\![W]\!]_{\{\!\{\Delta\}\!\}_\rho}$ *y* $\{\!\{\Gamma\}\!\}_\rho \leq \{\!\{\Delta\}\!\}_\rho$.

Por otro lado, la adecuación computacional asegura que una expresión se reduce a un valor si y sólo si su denotación no es indefinida.

Teorema 9 (Adecuación de ENS.) *Sea* $E \in \mathit{EExp} \cup \mathcal{C}\mathit{han}$, $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle \in \mathit{EHeap}$, *y* $\theta \notin \mathit{dom}(\overline{\Gamma})$. $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, *si y sólo si* $[\![E]\!]_{\{\!\{\Gamma\}\!\}_\rho} \neq \bot$.

Las demostraciones de estos dos últimos teoremas se realizan siguiendo los pasos dados en [Lau93], lo que nos ha llevado a los trabajos expuestos en la Sección 3.1.

Resumen de Resultados.

1. Definición de una semántica operacional distribuida para n procesadores.
2. Definición de una semántica operacional distribuida limitada a un único procesador.
3. Semántica natural extendida para la aplicación paralela.
4. Consistencia entre la semántica natural y su extensión.
5. Equivalencia entre la semántica de un procesador y la extensión paralela.
6. Corrección de la semántica natural extendida con respecto a una semántica denotacional extendida.

3.3 Trabajos relacionados

Como ya dijimos en el Capítulo 1, los resultados de Launchbury [Lau93] han tenido gran repercusión en distintas investigaciones. No hemos sido las únicas que se han percatado de la importancia de formalizarlos. Breitner en [Bre13, Bre14] ha desarrollado un estudio muy relacionado con el nuestro ya que su objetivo también es comprobar la corrección y adecuación computacional de la semántica natural de Launchbury, utilizando para ello un asistente de demostración.

Para demostrar la corrección de la semántica natural, Breitner propone dos métodos en [Bre14]. En el primero introduce una semántica equivalente a la de Launchbury en la que los juicios son de la forma $\Gamma : \Gamma' \Downarrow \Delta : \Delta'$, dónde Γ' y Δ' son conjuntos ordenados de ligaduras. Estos conjuntos no son arbitrarios, sino sendas pilas de ligaduras que reflejan cómo se va demandando la evaluación de las expresiones. De esta forma se mantiene más información en los juicios que la que se tenía con la semántica original de Launchbury. Por ejemplo, al evaluar una variable la ligadura correspondiente no desaparece, sino que pasa al *heap* Γ'. La idea es similar a la expuesta en la Sección 3.2, en la que se extienden los *heaps* para retener la información referente a las ligaduras bloqueadas. Tal y como explica el autor, en el caso de un lenguaje como el que utiliza Launchbury, y adecuándolo a sus reglas semánticas, solo se modificará en cada regla de derivación la primera ligadura de la pila actualizándola a un valor. Aún así, Breitner considera esta notación más natural y, además, podría adecuarse posteriormente a casos en los que se modifique la pila, por ejemplo, y tal y como él indica, mediante una recolección de basura. El segundo método consiste en modificar la semántica denotacional redefiniendo la función que relaciona los *heaps* con los entornos explicada en la Sección 2.4, de forma que el operador para la menor cota superior es reemplazado por una actualización.

Breitner, en [Bre13], ha formalizado en el asistente de demostración Isabelle parte de nuestro trabajo relativo a las semánticas denotacionales expuesto en la Sección 3.1.1 referente a la publicación P1. En dicho trabajo prueba la adecuación computacional de la semántica natural de Launchbury pero sin utilizar la semántica alternativa. Launchbury propuso esta nueva versión de modo que los *heaps* de la semántica operacional alternativa se correspondieran con los entornos de la semántica denotacional. La propuesta de Breitner solventa estos problemas desde el lado denotacional a través de una semántica denotacional de recursos alternativa, evitando tener que establecer la equivalencia entre la semántica operacional de Launchbury original y su versión alternativa.

Las técnicas nominales utilizadas por Breitner en la formalización en Isabelle tienen su correspondencia con la representación localmente sin nombres que hemos utilizado en nuestro estudio, ya que como explica el autor en [Bre14], en ambos casos, al realizar las demostraciones, se tienen en cuenta las variables que aparecen en los *heaps* para evitar la

captura de variables.

En la literatura pueden encontrarse diferentes métodos para establecer equivalencias entre expresiones de un λ-cálculo. Queremos destacar aquí los estudios de Haeri [Hae09, Hae13] por estar relacionados con esta tesis en ciertos aspectos. El lenguaje utilizado en [Hae09] es similar al λ-cálculo de Launchbury [Lau93] pero extendido con el operador `seq`. Además Haeri dota a este lenguaje de una semántica basada en la semántica de paso largo de Launchbury pero con una diferencia con respecto a la regla para las declaraciones locales. Al igual que Launchbury, introduce en el *heap* el conjunto de variables locales para obtener el valor de la expresión principal de la declaración, pero una vez obtenido el valor elimina del *heap* dichas variables. Esto da lugar a ciertas propiedades de los *heaps* inicial y final de una derivación, ya que sus dominios serán iguales y la diferencia entre ellos vendrá dada solo por la actualización de las ligaduras. Bajo esta semántica se introducen tres tipos de equivalencia entre expresiones:

- *Equivalencia de valores*: dos expresiones son equivalentes si al evaluarlas en el mismo contexto inicial producen el mismo valor;
- *Equivalencia de* heaps: dos expresiones son equivalentes si al evaluarlas en el mismo contexto inicial producen el mismo contexto final;
- *Equivalencia estricta*: dos expresiones son equivalentes si al evaluarlas en el mismo contexto inicial producen el mismo valor y el mismo *heap*.

Aunque estas equivalencias dan lugar a propiedades interesantes, no hemos podido aplicarlas en nuestro caso por dos razones: en primer lugar, porque ello requeriría modificar las reglas semánticas dadas por Launchbury, y nuestro objeto de estudio es la equivalencia de dichas reglas tal y cómo las propuso el autor; en segundo lugar, porque las equivalencias mencionadas anteriormente relacionan distintas expresiones evaluadas con la misma semántica.

La misma idea, en cuanto a las declaraciones locales se refiere, aparece en otro trabajo del mismo autor [Hae13], correspondiente a un modelo distribuido en el que no aparece el operador `seq`, pero se amplia el lenguaje con el operador #, que representa la aplicación estricta en la que tanto la función como el argumento son evaluados en el mismo contexto antes de realizar la aplicación. La diferencia fundamental con nuestro modelo distribuido es que no incorpora paralelismo.

Para lidiar con la α-conversión existen otras alternativas a la notación de de Bruijn y a la representación localmente sin nombres explicadas en la Sección 2.6. La lógica nominal [Pit03, GP02] es una de las más usadas en la actualidad. La base sobre la que se fundamenta esta lógica es que los predicados que describen propiedades sintácticas son equivariantes, en el sentido de que su validez es invariante bajo el intercambio (*swapping*) de nombres [Pit03].

La lógica nominal no sólo presenta ventajas con respecto a los problemas planteados por la α-conversión, además está implementada en Isabelle, por lo que puede utilizarse este asistente de demostración para estudiar propiedades relativas a ella. En nuestro caso, optamos por la representación localmente sin nombres porque, cuando iniciamos el proceso de implementación en Isabelle, la recursividad mutua de las declaraciones locales podía presentar problemas, y porque el trabajo realizado por Chargueráud [Cha11] e implementado en Coq se ajustaba bastante bien a nuestras necesidades.

El trabajo de Cimini et al. [CMRG12] sobre semánticas operacionales estructurales utiliza las técnicas de la lógica nominal. Dicho estudio trabaja con la noción de bisimilaridad. Basándose en las técnicas nominales introducidas por Pitts, Gabbay y Urban [GP99, UPG04], se desarrolla un entorno de trabajo para cálculos nominales llamado

Nominal SOS (*Nominal Structural Operational Semantics*), que se aplica a la noción de bisimilaridad nominal. Posteriormente, utilizando Nominal SOS, formulan el λ-cálculo perezoso y comprueban que coincide con el λ-cálculo original. Además, demuestran que la noción de bisimilaridad nominal coincide con la noción de bisimilaridad aplicativa de Abramsky explicada en la Sección 2.3.3.

3.4 Conclusiones

A lo largo de este capítulo se han ido mostrando los resultados que hemos obtenido en esta tesis. Primero se han relacionado dos semánticas denotacionales, una estándar y otra de recursos. Esta relación se establece mediante la relación de similaridad entre los valores de los dominios de definición de ambas semánticas. Mientras que la solución de la ecuación de dominios estándar, $D = [D \to D]_\perp$, ya estaba construida, para relacionar dichos valores, ha sido necesario construir previamente la solución de la nueva ecuación de dominios $E = [[C \to E] \to [C \to E]]_\perp$, siguiendo los pasos de Abramsky comentados en 2.3.2.

A continuación, para relacionar la semántica natural de Launchbury con su versión alternativa hemos introducido dos semánticas intermedias: una semántica con indirecciones, y otra sin actualización de ligaduras. Hemos demostrado la equivalencia entre la versión alternativa y la versión sin actualización, definiendo un preorden entre *heaps*, $\succsim_I$. Para establecer esta relación de indirecciones hemos definido previamente una serie de equivalencias entre términos y *heaps*. Además, hemos realizado este estudio utilizando la notación *localmente sin nombres*, para lo cual hemos tenido que extender algunas de las definiciones y propiedades ya existentes para esta notación, para así poder trabajar con un λ-cálculo con declaraciones locales.

Finalmente, hemos considerado un modelo distribuido de un lenguaje con aplicaciones paralelas. Para demostrar la corrección con respecto de una semántica denotacional extendida que dota de significado a las nuevas expresiones, hemos introducido dos semánticas intermedias: una semántica de 1-procesador y una extensión de la semántica natural de Launchbury. Hemos demostrado la consistencia entre la semántica natural y su extensión; la equivalencia entre la semántica de 1-procesador y la extensión de la semántica natural; y la corrección de la semántica natural extendida con respecto de la extensión de la semántica denotacional.

A continuación exponemos todos los resultados obtenidos que se han ido mostrando a lo largo de este capítulo.

- Construcción de la solución inicial de la ecuación $E = [[C \to E] \to [C \to E]]_\perp$, donde C representa el dominio de los recursos.
- Definición de una relación de *similaridad* entre los valores del dominio E y los valores del espacio de funciones estándar $D = [D \to D]_\perp$.
- Aplicación del resultado anterior para demostrar la equivalencia entre la semántica denotacional estándar y la semántica con recursos para un λ-cálculo perezoso.
- Representación localmente sin nombres del λ-cálculo extendido con declaraciones locales recursivas.
- Versión localmente sin nombres de las reglas de la semántica natural de Launchbury y su versión alternativa.

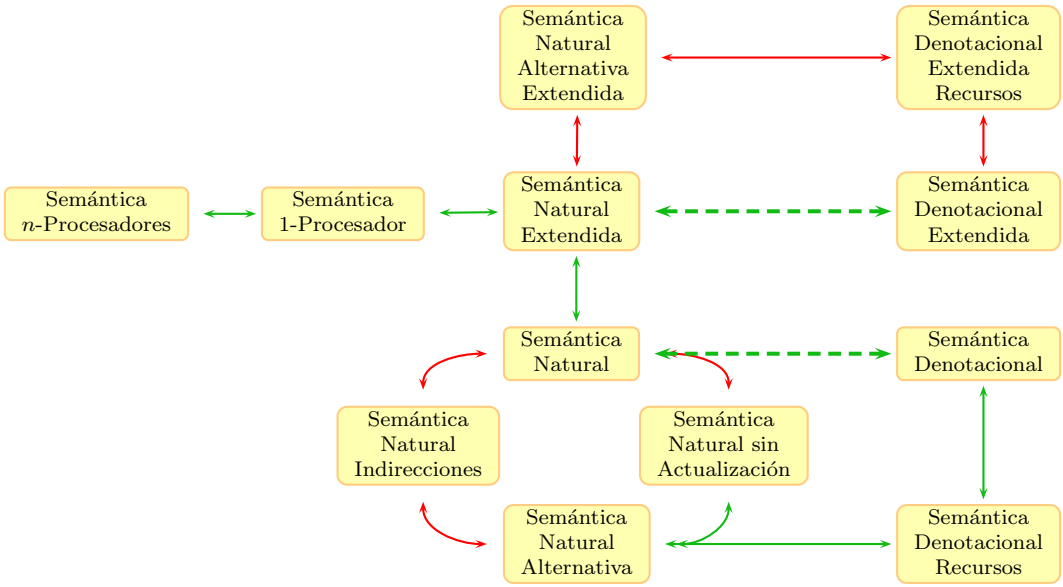
- Propiedades de los sistemas de reducción (lemas de regularidad, introducción y renombramiento).
- Relación de equivalencia entre *heaps* que definen las mismas variables libres pero cuyas clausuras pueden diferir en las variables libres indefinidas.
- Preorden que relaciona dos *heaps* cuando el primero puede transformarse en el segundo mediante la eliminación de indirecciones ($\succsim_I$).
- Extensión del preorden para pares (*heap*, término).
- Equivalencia de ANS y NNS.
- Definición de una semántica operacional distribuida para n procesadores.
- Definición de una semántica operacional distribuida limitada a un único procesador.
- Semántica natural extendida para la aplicación paralela.
- Consistencia entre la semántica natural y su extensión.
- Equivalencia entre la semántica de un procesador y la extensión paralela.
- Corrección de la semántica natural extendida con respecto a una semántica denotacional extendida.

Capítulo 4

¿Qué queda por hacer?

Cómo ya se comentó en el Capítulo 1, el objetivo inspirador de esta tesis era estudiar la equivalencia entre distintas semánticas de un modelo distribuido. El estudio de distintas semánticas operacionales y denotacionales y de las relaciones que hay entre ellas ha abierto muchos caminos interesantes en los que seguir trabajando.

Este capítulo se centra en las líneas de trabajo que se barajan a corto y medio plazo. Por un lado, en el siguiente esquema se muestran en color rojo las equivalencias que queremos completar para terminar el presente estudio:

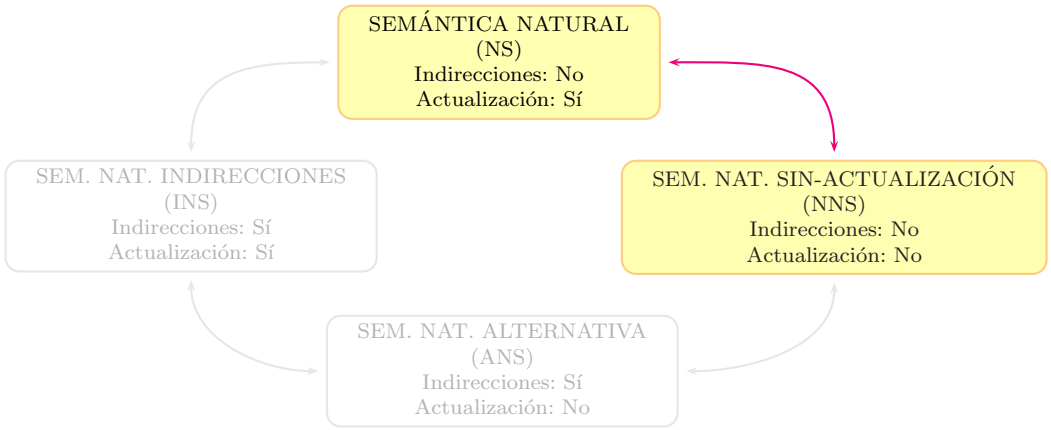


Por otro lado, consideramos interesante implementar todo este estudio en un asistente de demostración, para así obtener demostraciones formales con garantía de corrección.

A continuación, se exponen con más detalle estas líneas de investigación: la Sección 4.1 muestra una investigación en desarrollo con la que se concluirá la demostración de equivalencia entre las dos semánticas operacionales presentadas por Launchbury; la Sección 4.2 muestra una via alternativa para establecer la equivalencia entre la semántica natural de Launchbury y su versión alternativa; a continuación, en la Sección 4.3, se comenta la posible extensión de los resultados obtenidos y en progreso al modelo distribuido; concluimos con la Sección 4.4, en la que se expone la implementación de algunos de los resultados en el asistente de demostración Coq.

4.1 Equivalencia NS y NNS (Publicación WP1)

La primera tarea que queremos completar es la demostración de la equivalencia entre semántica natural de Launchbury y la versión alternativa. Tal y como muestra el siguiente diagrama, para concluir esta equivalencia queda por establecer la equivalencia entre la semántica natural (NS) y la versión sin actualización (NNS):



Este trabajo ya ha sido comenzado y está en pleno desarrollo, pero no ha sido incluido en el cuerpo principal de la tesis por estar aún incompleto. A continuación explicamos escuetamente los pasos seguidos hasta el momento y que pueden verse con más detalle en el trabajo en progreso WP1, que aparece en el Apéndice B. El siguiente ejemplo muestra las diferencias entre los *heaps* finales producidos al evaluar una expresión con las dos semánticas involucradas.

Ejemplo 9 *Considérese el término*

$$t \equiv \begin{array}{l} \texttt{let abs (bvar 0 0), } s\texttt{, app (bvar 0 0) (bvar 0 1)} \\ \texttt{in app (app (app (bvar 0 1) (bvar 0 0)) (bvar 0 0)) (bvar 0 2)} \end{array}$$

donde

$$s \equiv \begin{array}{l} \texttt{let abs (bvar 0 0), abs (bvar 0 0), app (bvar 0 0) (bvar 0 1)} \\ \texttt{in app (bvar 0 0) (bvar 0 1)} \end{array}$$

Al evaluar dicha expresión con NS ($\Downarrow$) y con NNS ($\Downarrow^N$) en el contexto de un heap *vacío, se obtienen respectivamente los siguientes resultados:*

$\{\,\} : t \Downarrow$

$x_0 \mapsto \texttt{abs (bvar 0 0)},$
$x_1 \mapsto \texttt{abs (bvar 0 0)},$
$x_2 \mapsto \texttt{abs (bvar 0 0)},$
$y_0 \mapsto \texttt{abs (bvar 0 0)},$
$y_1 \mapsto \texttt{abs (bvar 0 0)},$
$y_2 \mapsto \texttt{app (fvar } y_0\texttt{) (fvar } y_1\texttt{)}$

$\{\,\} : t \Downarrow^N$

$x_0 \mapsto \texttt{abs (bvar 0 0)},$
$x_1 \mapsto s,$
$x_2 \mapsto \texttt{app (fvar } x_0\texttt{) (fvar } x_1\texttt{)},$
$y_0 \mapsto \texttt{abs (bvar 0 0)},$
$y_1 \mapsto \texttt{abs (bvar 0 0)},$
$y_2 \mapsto \texttt{app (fvar } y_0\texttt{) (fvar } y_1\texttt{)},$
$z_0 \mapsto \texttt{abs (bvar 0 0)},$
$z_1 \mapsto \texttt{abs (bvar 0 0)},$
$z_2 \mapsto \texttt{app (fvar } z_0\texttt{) (fvar } z_1\texttt{)}$

$w \equiv \texttt{abs (bvar 0 0)}$

□

Por un lado, debido a la no actualización, con NNS se repiten los cómputos de evaluación de una variable cada vez que esta es demandada. De este modo, se incorporarán al *heap* varias veces las mismas ligaduras, aúnque con nombres distintos. Por otro lado, las ligaduras que ya han sido evaluadas aparecen como tales en el *heap* obtenido con NS, pero aparecen sin evaluar en el obtenido con NNS. Por ello, la relación entre estos *heaps* se establece en dos pasos: primero se eliminan aquellos grupos de ligaduras que sean "equivalentes"; y después se comprueba que las variables que aparezcan ligadas a términos sin evaluar en NNS pero evaluados en NS en realidad evalúan al mismo valor.

Para detectar los grupos que nos interesan para la primera fase, definimos una equivalencia de contextos, $\approx^{(\overline{x},\overline{y})}$, en la que dos términos t y t' son equivalentes en los contextos $\overline{x}$ e $\overline{y}$ si los términos son iguales al cerrarlos en sus respectivos contextos. Utilizamos esta noción para eliminar grupos equivalentes. Así, dos pares $(\Gamma : t)$ y $(\Gamma' : t')$ estarán relacionados si al eliminar del primero un grupo equivalente a otro, el par obtenido (*heap*, término) está relacionado con $(\Gamma' : t')$, tal y cómo se indica en la siguiente definición:

Definición 1 *Dados dos pares* (heap, *término*) $(\Gamma : t)$ *y* $(\Gamma' : t')$*, se dice que* $(\Gamma : t)$ *está relacionado con* $(\Gamma' : t')$ *mediante* grupos*, lo que denotamos por* $(\Gamma : t) \succsim_G (\Gamma : t)$*, si ello puede deducirse aplicando las siguientes reglas:*

$$\frac{}{(\Gamma : t) \succsim_G (\Gamma : t)} \qquad \frac{\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s} \qquad \overline{x} \cap \overline{y} = \emptyset \qquad ((\Gamma, \overline{x} \mapsto \overline{t}) : t)[\overline{x}/\overline{y}] \succsim_G (\Gamma' : t')}{((\Gamma, \overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{s}) : t) \succsim_G (\Gamma' : t')}$$

En la definición anterior, $\overline{x} \mapsto \overline{t}$ e $\overline{y} \mapsto \overline{s}$ representan dos grupos de ligaduras equivalentes ($\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s}$). Aplicamos esta definición a los *heap* del Ejemplo 9:

Ejemplo 10 *Veamos qué ocurre al eliminar el grupo* $\overline{z} = [z_0, z_1, z_2]$ *del* heap *obtenido con NNS en el Ejemplo 9.*

$x_0 \mapsto$ `abs` (`bvar` 0 0),
$x_1 \mapsto s$,
$x_2 \mapsto$ `app` (`fvar` x_0) (`fvar` x_1),
$y_0 \mapsto$ `abs` (`bvar` 0 0),
$y_1 \mapsto$ `abs` (`bvar` 0 0),
$y_2 \mapsto$ `app` (`fvar` y_0) (`fvar` y_1),
$z_0 \mapsto$ `abs` (`bvar` 0 0),
$z_1 \mapsto$ `abs` (`bvar` 0 0),
$z_2 \mapsto$ `app` (`fvar` z_0) (`fvar` z_1)

$\succsim_G$

$x_0 \mapsto$ `abs` (`bvar` 0 0),
$x_1 \mapsto s$,
$x_2 \mapsto$ `app` (`fvar` x_0) (`fvar` x_1),
$y_0 \mapsto$ `abs` (`bvar` 0 0),
$y_1 \mapsto$ `abs` (`bvar` 0 0),
$y_2 \mapsto$ `app` (`fvar` y_0) (`fvar` y_1)

Nótese que el heap *obtenido tras eliminar el grupo tiene el mismo dominio que el* heap *obtenido con NS en el Ejemplo 9.* □

Una vez eliminados todos los grupos equivalentes, los *heaps* obtenidos tienen el mismo domino. Definimos entonces una relación de actualización (*update*) sobre *heaps*, de forma que dos *heaps* están relacionados si y sólo si tienen el mismo dominio y para cada variable no evaluada en el primer *heap* que esté evaluada en el segundo, el valor producido al evaluar su término correspondiente en un contexto dado produce un valor "equivalente" al valor ligado a la misma variable en el segundo *heap*.

Definición 2 *Sean* $\Gamma, \Gamma', \Delta \in \mathit{LNHeap}$*. Se dice que* Γ *está relacionado con* Γ' *mediante* update en el contexto de Δ*, y lo denotamos por* $\Gamma \sim_U^{\Delta} \Gamma'$*, si ello puede deducirse aplicando las siguientes reglas:*

$$\frac{}{\Gamma \sim_U^{\Delta} \Gamma} \qquad \frac{\Gamma \sim_U^{\Delta} \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma, x \mapsto t) \sim_U^{\Delta} (\Gamma', x \mapsto w')}$$

Nos interesa el caso en el que el contexto de evaluación es el *heap* sin actualizar. Por ello, cuando no se especifica el contexto nos referimos siempre al primer *heap*; es decir, $\Gamma \sim_U \Gamma'$, si $\Gamma \sim_U^{\Gamma} \Gamma'$, y entonces decimos que Γ está relacionado mediante *update* con Γ'.

Esta definición la extendemos a pares (*heap*, términos) en la forma siguiente:

$$\frac{\Gamma \sim_U^{\Delta} \Gamma'}{(\Gamma : t) \sim_U^{\Delta} (\Gamma' : t)} \qquad \frac{\Gamma \sim_U^{\Delta} \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma : t) \sim_U^{\Delta} (\Gamma' : w')}$$

Combinando las relaciones de grupo y actualización se obiene la relación de grupo-actualización sobre pares (*heap*, término), $\succsim_{GU}$:

$$\frac{(\Gamma : t) \succsim_G (\Delta : s) \quad (\Delta : s) \sim_U (\Gamma' : t') \quad \texttt{ok}\ \Gamma \quad \texttt{ok}\ \Gamma' \quad \texttt{lc}\ t \quad \texttt{lc}\ t'}{(\Gamma : t) \succsim_{GU} (\Gamma' : t')}$$

Para completar este trabajo queda por demostrar el teorema de equivalencia, que indica que partiendo del mismo par (*heap*, término) los pares obtenidos al evaluar entre NS y NNS están relacionados mediante la relación de grupo-actualización. Prevemos que, a semejanza de lo ocurrido para la equivalencia con NNS y ANS (Sección 3.1.2), será necesaria una generalización para poder aplicar la inducción sobre las reglas semánticas, y sin duda varios lemas auxiliares.

4.2 Equivalencias entre NS y INS y entre INS y ANS

La equivalencia entre NS y ANS quedaría probada con el resultado principal de la publicación P3 y el resultado que queda por probar de la sección anterior. Aún así, consideramos interesante cerrar el diagrama entre las cuatro semánticas operacionales, y demostrar así la equivalencia entre la semántica natural y la alternativa tomando el camino en el que participa la semántica natural con indirecciones (INS), sin tener en cuenta los resultados obtenidos al considerar la semántica sin actualización (NNS).

Al igual que en la otra parte del diagrama, haremos el estudio en dos pasos. Por un lado, desarrollamos la equivalencia entre la semántica natural con indirecciones y la semántica alternativa. Siendo la *regla variable* la única regla semántica en la que difieren, prevemos que sea un estudio muy similar al que estamos realizando entre la semántica natural y su versión sin actualización. Por otro lado, esperamos que la demostración de la equivalencia entre la semántica natural y su versión con indirecciones sea parecida a la ya realizada para la semántica alternativa y aquella que no tiene actualizaciones (Sección 3.1.2). Sin embargo, se ha de tener en cuenta que, al haber actualización de clausuras, algunas de las indirecciones podrían perder su forma. Por ello, tendremos que estudiar no sólo cómo eliminar indirecciones, sino también aquellas ligaduras que sean "redundantes" por estar ligadas a los mismos valores.

4.3 Extensión al modelo distribuido

Una vez finalizada la parte del cálculo secuencial, pretendemos extender los resultados al modelo distribuido. Al efecto será neceserario definir una versión alternativa de la semántica natural extendida (ENS) analizando si es necesario incluir indirecciones cuando se trabaja con aplicaciones paralelas. Igualmente habrá que definir una semántica denotacional de recursos para esta extensión. Para demostrar la corrección y adecuación de estas habrá que demostrar primero la equivalencia entre las versiones extendidas de las semánticas denotacionales utilizando una relación de "similaridad", para después demostrar la equivalencia entre las semánticas operacionales extendidas. Posiblemente sea necesario definir unas semánticas intermedias que corresponderían a las versiones extendidas de la semántica natural de indirecciones y de no actualización. Previo al estudio de las equivalencias, será necesario expresar el lenguaje y las reglas semánticas con la notación localmente sin nombres.

4.4 Implementación en Coq (Publicación WP2)

Aunque sigue habiendo detractores, son bien conocidas las ventajas que ofrecen hoy en día los demostradores automáticos y los asistentes de demostración cuando nos referimos a la fiabilidad de las demostraciones. Desde un primer momento tuvimos claro que utilizar alguno de los asistentes de demostración existentes (algunos de los más importantes han sido nombrados en la Sección 2.7) era fundamental en este estudio.

Hace años empezamos trabajando con Isabelle, pero desafortunadamente en aquellos momentos el paquete de Nominal Isabelle estaba en sus inicios y la recursión mutua de las declaraciones locales del lenguaje con el que trabajamos podía dar problemas. Sin el uso del citado paquete, había que trabajar utilizando la notación de de Bruijn y ya se han comentado en la Sección 2.6.1 las desventajas de esta notación.

El hecho de que Coq aceptara en esos momentos definiciones de tipo inductivo, lo que nos permitiría trabajar con las declaraciones locales recursivas, y que ya hubiera trabajos en Coq utilizando la notación localmente sin nombres [Cha11], nos hizo inclinarnos a favor de este segundo asistente.

Aunque el lenguaje que utilizamos tiene las aplicaciones restringidas a variables (Figura 3.2), al utilizar Coq nos resultó más sencillo trabajar con aplicaciones de términos a términos. Para ampliar el trabajo de Chargéraud [Cha11], empezamos ampliando la definición del lenguaje con declaraciones locales recursivas, que son representadas mediante una lista de términos:

```
Inductive trm :  Type :=
                 | t_bvar : nat− > nat− > trm
                 | t_fvar : var− > trm
                 | t_abs : trm− > trm
                 | t_app : trm− > trm− > trm
                 | t_let : L_trm− > trm− > trm
           with  L_trm :=
                 | nil_Lt : L_trm
                 | cons_Lt : trm− > L_trm− > L_trm.
```

Al comprobar la definición del principio de inducción resultó que la inducción estándar que define Coq no se transmite a las listas de términos. A continuación mostramos esta inducción, en la que se observa que al llegar a las declaraciones locales la propiedad solamente se comprueba en el término principal y no en los términos que representan las

declaraciones locales:

```
trm_ind :  forall P : trm -> Prop,
           (forall n n0 : nat, P (trm_bvar n n0)) ->
           (forall v : var, P (trm_fvar v)) ->
           (forall t : trm, P t -> P (trm_abs t)) ->
           (forall t : trm, P t -> forall t0 : trm, P t0 -> P (trm_app t t0)) ->
           (forall (l : L_trm) (t : trm), P t -> P (trm_let l t)) ->
           forall t : trm, P t
```

Por ello, y puesto que en nuestro caso es imprescindible que las declaraciones locales también verifiquen la inducción, redefinimos el principio de inducción con dos propiedades (P para los términos y P0 para las listas) que son mutuamente recursivas:

```
trm_ind2  forall (P : trm -> Prop) (P0 : L_trm -> Prop),
          (forall n n0 : nat, P (trm_bvar n n0)) ->
          (forall v : var, P (trm_fvar v)) ->
          (forall t : trm, P t -> P (trm_abst)) ->
          (forall t : trm, P t -> forall t0 : trm, P t0 -> P (trm_app t t0)) ->
          (forall l : L_trm, P0 l -> forall t : trm, P t -> P (trm_let l t)) ->
          P0 nil_Ltrm ->
          (forall t : trm, P t -> forall l : L_trm, P0 l -> P0 (cons_Ltrm t l)) ->
          forall t : trm, P t
```

Una vez subsanados los problemas provocados por la definición del principio de inducción, extendimos las definiciones de apertura, cierre, substitución y variables libres de un término. Puesto que la definición de términos es una definición mutuamente recursiva en la que para construir un término se requieren listas de términos, y para construir las listas se requieren términos, todas estás definiciones hay que hacerlas simultánemente para los términos y las listas. Aquí mostramos como ejemplo sólo la de clausura en un nivel k:

```
Fixpoint close_rec (k : nat) (vs : list var) (t : trm) {struct t} : trm :=
    match t with
    | t_bvar i j => t_bvar i j
    | t_fvar x => if (search_var x vs)
                  then (t_bvar k (pos_elem x vs 0))
                  else (t_fvar x)
    | t_abs t1 => t_abs (close_rec (S k) vs t1)
    | t_app t1 t2 => t_app (close_rec k vs t1) (close_rec k vs t2)
    | t_let ts t => t_let (close_L_rec (S k) vs ts)(close_rec (S k) vs t)
    end
with close_L_rec (k : nat) (vs : list var) (ts : L_trm) {struct ts} : L_trm :=
    match ts with
    | nil_Lt => nil_Lt
    | cons_Lt t ts => cons_Lt (close_rec k vs t) (close_L_rec k vs ts)
    end.
```

Igualmente, el predicado de clausura local ha de hacerse simultáneamente sobre términos y listas de términos:

```
Inductive lc : trm -> Prop :=
    | lc_var : forall x, lc (t_fvar x)
    | lc_app : forall t1 t2, lc t1 -> lc t2 -> lc (t_app t1 t2)
    | lc_abs : forall L t, (forall x, x ∉ L -> lc (open t (cons x nil))) -> lc (t_abst)
    | lc_let : forall L t ts, (forall xs, xs ∉ L -> (lc_list (opens (cons_Lt t ts) xs)))
                          -> lc (t_let ts t)
with lc_list : L_trm -> Prop :=
    | lc_list_nil : lc_list (nil_Lt)
    | lc_list_cons : forall t ts, lc t -> lc_list ts -> lc_list(cons_Lt t ts).
```

Posteriormente avanzamos definiendo las ligaduras, los *heaps*, funciones sobre los *heaps* (como el dominio, los nombres, la sustitución de nombres y el predicado de *heaps* bien definidos comentados en la Sección 3.1.2). Aún así, estas definiciones tienen que ser revisadas ya que parece que no son las más adecuadas para la definición de las reglas semánticas y las posteriores demostraciones. La definición de *heap* se ha dado como conjunto de ligaduras, pero algunos expertos en Coq nos han indicado que quizá sería más conveniente hacerlo mediante una función parcial. Por ello habrá que indagar las diferencias entre ambas definiciones y las ventajas e inconvenientes que pueda tener cada método.

Esta parte de la investigación ha sido aplazada para ser retomada posteriormente. Trabajar con asistentes de demostración es complejo y demostraciones que en lápiz y papel parecen sencillas, pueden convertirse en algo muy complicado. Era imprescindible dedicarle un tiempo del que no disponíamos en el momento en el que se comenzó a elaborar este trabajo y que esperamos poder tener en el futuro. Por ello, al igual que la publicación WP1 descrita en la Sección 4.1, WP2 no forma parte del núcleo de la tesis.

Ciertamente aún queda camino por recorrer pero lo ya hecho y expuesto en el Capítulo 3 tiene la suficiente entidad (unidad, dificultad, volumen, claridad y representatividad) como para dar lugar a la Tesis doctoral que presentamos aquí.

Part II

Summary of the Research

Chapter 1

What, why and how?

When several semantics are defined for a programming language, these semantics should be equivalent, in the sense that equivalent meanings are given for each program written in the language. Hidalgo-Herrero, in her PhD thesis [Hid04], defined an operational and a denotational semantics for `Jauja`, a simplification of `Eden` [BLOP96]. This language has two differentiated parts:

- a lazy λ-calculus; and
- coordination expressions.

The initial purpose of this thesis was to prove the equivalence between the semantics defined by Hidalgo-Herrero. But this goal turned out to be too ambitious, since the coordination expressions of `Jauja` complicated considerably the task of proving the equivalence between a small-step operational semantics and a continuation-based denotational semantics. Therefore, the final target of this thesis is to give the first steps in proving the desired equivalence.

For a start, we have based our work on Launchbury's ideas [Lau93] to prove the equivalence between a big step natural semantics and a standard denotational semantics defined for a λ-calculus extended with local declarations. We have extended the λ-calculus used by Launchbury with a parallel application. This expression gives rise to the creation of new processes and the communication between them. In order to define the meaning of the new expression and the new identifiers (for channels) we have extended the natural semantics. These extensions had to be consistent with Launchbury's and with Hidalgo-Herrero's definitions.

We can differentiate two parts in this thesis: one is dedicated to a distributed model formed by several processes that interact between them; the other part focuses on a one processor model.

Paradoxically, the study of the distributed model has led to a deeper examination of the one processor model, where we have two parts: in the first one we studied the characteristics and relations between denotational semantics; in the second one we analyzed the properties and relations of several operational semantics.

During the research we have found a number of problems that we have solved. Some parts of this thesis are the consequence of the absence of detailed proofs in [Lau93]. Although Launchbury gave some intuitive ideas, its development is more complex than expected and rule induction is insufficient. Since several works for some results [BKT00, HO02, NH09, Ses97, vEdM07] are based on Launchbury's study, we considered very important to formalize those results.

The notation for representing the expressions of a language can facilitate or complicate the formalization of proofs. Working with a λ-calculus usually gives problems related with the α-conversion. Several techniques have been developed to avoid them: the de Bruijn notation [dB72], the locally nameless representation [Cha11] or the techniques of the nominal logic [Pit13]. We have worked with the second option in some of the articles that form this thesis.

Summarizing, we have studied different semantics and techniques for the representation of terms of a λ-calculus with local declarations in order to establish the equivalence between some semantics of a distributed model.

1.1 Objectives

The main purpose of this thesis is:

- to begin with the proof of the equivalence between the semantics for `Jauja` given in [Hid04].

In order to achieve this, we focus on the following targets:

- to extend the λ-calculus with a parallel application, i.e., to introduce explicit parallelism;
- to define for this extension different operational and denotational semantic models (for one and for several processors);
- to study the relation between the defined semantic models, that is, to formalize the equivalence between the semantics defined in the previous step;
- to formalize the missing proofs in [Lau93], particularly the equivalence between a standard denotational semantics and a resourced denotational semantics, and between Launchbury's natural semantics and its alternative version.

1.2 Summary

This thesis is composed by a collection of publications. In order to understand the relation between these publications and to facilitate its reading, we have written this introduction and three additional chapters. We explain in Chapter 2 some preliminary concepts useful to understand the material presented in the publications. Chapter 3 is dedicated to the obtained results. All these results are detailed in the publications, therefore the purpose of this chapter is only to give an intuitive idea of them. Each section of the chapter is related with one or more publications, and this is explicity indicated. We also show in this chapter some of the related work. Future work is presented in Chapter 4. This is divided in four sections where we explain different future research lines and how much we have already done. Finally, Chapter 5 includes the main publications that compose the thesis:

- P1: *Relating function spaces to resourced function spaces* [SGHHOM11].
- P2: *A Locally Nameless Representation for a Natural Semantics for Lazy Evaluation* [SGHHOM12b].
- P3: *The Role of Indirections in Lazy Natural Semantics* [SGHHOM14b].

- P4: *An Operational Semantics for Distributed Lazy Evaluation* [SGHHOM10].

We have included two appendices. Appendix A contains two technical reports with extended versions of publications P2 and P3. In these extensions can be found detailed proofs of the results in the corresponding publications.

- TR1: *A locally nameless representation for a natural semantics for lazy evaluation* [SGHHOM12c].
- TR2: *The role of indirections in lazy natural semantics* [SGHHOM13].

To conclude, we show two works in progress in Appendix B. First steps have been completed on these works, but further development has been postponed for several reasons:

- WP1: *Launchbury's semantics revisited: On the equivalence of context-heap semantics* [SGHHOM14a].
- WP2: *A formalization in Coq of Launchbury's natural semantics for lazy evaluation* [SGHHOM12a].

Chapter 2

What was done?

We explain in this chapter some concepts developed by others and that are used in this thesis.

2.1 Programming languages

Every language is composed by a syntax and a semantics. The syntax shows how to build correct expressions, while the semantics gives meaning to them. This applies also to programming languages: the syntax determines how to build programs, and the semantics expresses how they behave when they are executed in a computer.

In the case of programming languages, formal semantics prevent ambiguities, i.e., each term has a unique meaning.

2.1.1 Functional Programming Languages

There are several programming paradigms. In the imperative one programmers define step by step how to solve a problem even when the procedure differs from the initial mathematical definition. Functional programming languages have a higher level of abstraction.

Let us consider a simple example. The mathematical definition to calculate the n^{th} power of a number is:

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x \cdot x^n \end{aligned}$$

In an imperative language as `C` the expression to represent the n^{th} power is:

```
int power(int x, int n);
{
    int i = 1;
    int result = 1;
    while (i <= n)
    {
        result  = result * x;
        i++;
    }
    return(result);
}
```

While in a functional language like `Haskell`, the definition is:

$$\begin{aligned} \text{power } x\ 0 &= 1 \\ \text{power } x\ n &= x * \text{power } x\ (n-1) \end{aligned}$$

Usually functional programming languages are considered to be less efficient than imperative ones. Barendregt and Barendsen explain in [BB00] that Von-Neumann architecture is based on the Turing machine. Imperative programming languages follow a sequence of instructions that are close to this architecture. However, functional programming languages are more efficient in reduction machines, which are created to execute these languages based in the λ-calculus. The lost of efficiency of functional programming languages is due to the fact that most computers have a Von-Neumann architecture. For this reason, lots of studies about functional languages have been devoted to develop efficient implementations of functional programming languages, and nowadays the execution time of these languages is competitive. Moreover, functional languages offer some advantages such as a higher level of abstraction, a shorter code, no side effects, easy debugging, concurrency, hot code deployment, natural recursion, etc. Thus, at the present time functional programming is not only restricted to the academic context. For instance `Haskell` is used in Intel, Deutsche Bank and even in Facebook and Google [has14a], while Erlang [erl14a] is used in Whatsapp [wha14], Facebook and T-Mobile [erl14b].

2.1.2 Evaluation strategies

Expressions in functional programming languages are evaluated by the reduction of subexpressions. Depending on the reduction order of *redexes* (reduction expressions), different evaluation strategies are obtained. They can be classified into two groups: the first one refers to those strategies where the evaluation of arguments is done before applying the function although they may not be required (eager evaluation); the second one refers to the strategies where the arguments are only evaluated if they are needed to continue with the computation (lazy evaluation).

In this thesis we use the following evaluation strategies (as defined in [Rea89]):

- *Call-by-value*: It is an eager strategy where the arguments are evaluated before the function's body;
- *Call-by-name*: It is an strategy of the second group where the argument (that has not been evaluated) is replaced in the function's body which is then evaluated. Although some expressions may be evaluated several times, if they are not required, they will not be evaluated at all;
- *Call-by-need*: It is a lazy strategy but more efficient than call-by-name since the value of an expression is stored and shared. Therefore, expressions are evaluated at most once.

2.1.3 Parallel functional languages

The development of parallel and distributed machines leads to the search of programming languages that facilitate parallel programming. Imperative languages are useful for this purpose, but synchronization and communication are treated in a very low level of abstraction. By contrast, functional languages present some advantages, as we have seen in Section 2.1.1.

Loogen [Loo99] classifies the parallelism in functional programming languages into three groups, depending on the allowance of the programmer to establish where parallelism has to be performed:

- *Implicit parallelism*: This is inherent in the reduction semantics. Independent *redexes* can be reduced in an arbitrary order. Therefore, they can be reduced in parallel. It is related to the automatic parallelization of functional programming languages.

- *Controlled parallelism*: The programmer adds some annotations indicating to the compiler where parallelism could be performed. High level parallel constructions are used, such as skeletons [Col89] or evaluation strategies [THLP98].

- *Explicit parallelism*: The programmer establishes which expressions must be computed in parallel. Some languages, such as `Haskell` [Pey03] or ML [MTH90], have extensions to deal with explicit process creation, communication and synchronization between processes.

The functional programming language `Haskell` [Pey03, has14b] has been the origin of several parallel and distributed versions [TLP03]. `Haskell` follows a lazy evaluation strategy (Section 2.1.2); since expressions are evaluated under demand, parallelism is restricted. The parallel versions of `Haskell` eliminate some laziness either by speculative computations, i.e., allowing the evaluation of non-demanded parts (for instance in `GpH` [THLP98] with the operator `par`), or by introducing strictness, i.e., forcing the evaluation of expressions before the result is needed (for instance the operator `seq` in `GpH` [THLP98]) .

2.1.4 The functional parallel language `Eden`

This thesis focuses on some properties of the kernel of the functional parallel language `Eden` [BLOP96, LOP05, ede14]. `Eden` is an extension of `Haskell` with some syntactic constructs for explicit process specification and creation, which provide enough control to implement parallel algorithms. It also introduces automatic communication via *streams*. The main characteristics of `Eden` are:

- *Process abstractions*: Expressions that define the general behavior of a process in a functional way.

- *Process creations*: Applications of the process abstractions to a group of expressions. These expressions produce the values of the input channels of the new process.

- *Communication between processes*: It is asynchronous and implicit, since the transmission of messages is not explicitly made by the programmer. Communications can be of just one value, or of several values through a *stream*.

The constructions of `Eden` also model reactive systems:

- *Dynamic channel creation*: This allows to break the hierarchy between processes. That is, in addition to communications between father and child processes, they are also between any pair of processes.

- *Non-determinism*: To model many-to-one communications, a special process abstraction merges several *streams* into one in a non-deterministic way.

2.2 Programming Language Semantics

In the preface of [Win93], Winskel explains that giving a formal semantics to a programming language lies in building a mathematical model. Formal semantics allow us to understand and to reason about programs behavior.

2.2.1 Formal semantics

There are different types of formal semantics. In this thesis we use two types of them:

- **Operational:** An operational semantics describes the meaning of a program in terms of how it is executed by an abstract machine. It focuses on both the final value and how this value is obtained. They are classified into two groups: *small step semantics*, which detail how computations are performed step by step, and *big step semantics or natural semantics*, which describe how the final value is directly obtained.

- **Denotational:** A denotational semantics gives meaning to programs through mathematical objects, which are called *denotations*. For its definition, the first task is to find some mathematical object representing the effect of a program. So that the semantics describes the function computed by the program, but it is not concerned in how this is done. The denotation of a term is obtained by the composition of the denotations of the subterms. This semantics has a higher level of abstraction than the operational semantics, and hence it is useful to study the equivalence between programs. The steps to define a denotational semantics are: to define a space of meanings, to give to each constant of the language a meaning in this space, to define for each constructor in the language a semantic function over the space of meanings, and finally, to define the main semantic function corresponding to the semantic value of each program.

When two (or more) formal semantics are defined for the same language, it must be proved that they are equivalent. When dealing with operational and denotational semantics, this equivalence is usually expressed in terms of correctness and computational adequacy:

- **Correctness:** An operational semantics is correct with respect to a denotational one, if the operational reduction preserves the denotational meaning of terms and only the meaning of the contexts of evaluation is modified.

- **Computational adequacy:** The computational adequacy of an operational semantics with respect to a denotational semantics establishes that whenever the denotation of some term is defined then there exists an operational reduction for this term.

In this thesis we work with several operational and denotational semantics of a functional programming language, and we study the relation between them.

2.3 Function spaces

As it is explained in [Win93], comparing programs that are written in different programming languages can be very complex if operational semantics are involved. The reason is that they deeply depend on the syntax of the language. However, denotational

semantics give a more abstract meaning to the expressions, whose values are in a function space.

Abramsky and Jung, in [AJ94], introduced the two main problems that give rise to the domain theory [Sco73]: Least fixpoints as meanings of recursive definitions and recursive domain equations. Abramsky [Abr91] also explained how domain theory, that was introduced by Scott, have been studied during the years, specially its applications to denotational semantics.

2.3.1 Basic concepts

We recall some important concepts of the domain theory. The following definitions are extracted from the book on formal semantics of programming languages [Win93].

A *partial order* is a set P on which a binary relation $\sqsubseteq$ has been defined; this relation must be reflexive, transitive and antisymetric. For a partial order $(P, \sqsubseteq)$ and subset $X \subseteq P$, $p \in P$ is an *upper bound of* X if and only if $\forall q \in X \;.\; q \sqsubseteq p$. Say $p \in P$ is a *least upper bound* (lub) of X ($\bigsqcup X$) if and only if p is an upper bound of X, and for all upper bounds q of X, $p \sqsubseteq q$.

A partial order $(P, \sqsubseteq)$ is a *complete partial order* (cpo) if any increasing chain of elements of P ($d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$) has a least upper bound ($\bigsqcup_n d_n$) in P. We say $(P, \sqsubseteq)$ is a cpo with bottom if it is a cpo with a least element, $\bot$.

A function $f : D \to E$ between cpos $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ is *monotonic* if and only if $\forall d, d' \in D \;.\; d \sqsubseteq_D d' \;\Rightarrow\; f(d) \sqsubseteq_E f(d')$. Such a function is *continuous* if and only if it is monotonic and for every chain $(d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots)$ in D, the lub of the images of the elements coincides with the image of the lub of the elements, that is, $\bigsqcup_n f(d_n) = f(\bigsqcup_n d_n)$.

Let $f : D \to D$ be a continuous function on a cpo $(D, \sqsubseteq_D)$. A *fixed point* of f is an element d of D such that $f(d) = d$.

Kleene fixed point theorem. *Let $f : D \to D$ be a continuous function on a cpo with bottom $(D, \sqsubseteq_D)$. Define $\mathtt{fix}(f) = \bigsqcup_n f^n(\bot)$. Then*

1. *$\mathtt{fix}(\mathtt{f})$ is a fixed point of f, i.e., $f(\mathtt{fix}(f)) = \mathtt{fix}(f)$;*
2. *If $f(d) = d$ then $\mathtt{fix}(f) \sqsubseteq d$.*

Thus $\mathtt{fix}(\mathtt{f})$ is the least fixed point of f.

Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be two cpos. The *function space* $[D \to E]$ is given by the elements of $\{f | f : D \to E \text{ is continuous}\}$ ordered point by point by $f \sqsubseteq g \stackrel{\text{def}}{=} \forall d \in D . f(d) \sqsubseteq g(d)$. Thus, the function space is a cpo and for each chain $f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_n \sqsubseteq \ldots$ the lub verifies that $(\bigsqcup_n f_n)d = \bigsqcup_n (f_n(d))$.

2.3.2 Construction of the initial solution

The pure λ-calculus does not correspond completely to the semantics of lazy functional languages. There are two types of elements: *convergent* elements and *divergent* ones. Convergent elements are those whose evaluation leads to functions from D in D (where D is a suitable domain for values), while divergent elements are those whose evaluation never ends. Abramsky in [Abr90] refers to this fact, and he establishes a theory based on *applicative transition systems* where these elements are distinguished. He introduces the

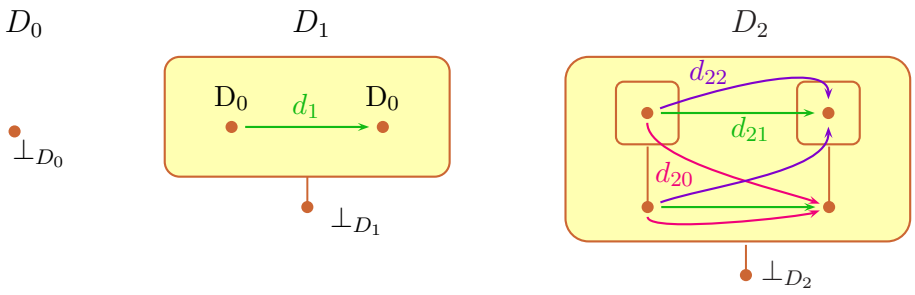


Figure 2.1: First three levels of the function space $[D \to D]_\perp$

domain equation $D = [D \to D]_\perp$, where $[D \to D]_\perp$ corresponds with the space of the continuous functions from D in D where a minimum element ($\perp$) has been added. This equation has a non-trivial initial solution that models the lazy functional languages. The construction of this initial solution is detailed in [AO93]. We resume here the main steps of this construction.

Let D and E be *cpo*s. The pair $\langle i, j\rangle$ is an *embedding* from D to E if i and j are continuous functions $D \overset{i}{\rightarrowtail} E \overset{j}{\twoheadrightarrow} D$ such that $i \circ j \sqsubseteq \mathtt{id}_E$ and $j \circ i = \mathtt{id}_D$, where $\overset{i}{\rightarrowtail}$ stands for an injection and $\overset{j}{\twoheadrightarrow}$ for a projection.

The construction of the function space is done by levels, with $D_0 \overset{\mathtt{def}}{=} \{\perp\}$ and $D_{n+1} \overset{\mathtt{def}}{=} [D_n \to D_n]_\perp$. For each consecutive level we can construct the continuous functions $D_n \overset{i_n}{\rightarrowtail} D_{n+1} \overset{j_n}{\twoheadrightarrow} D_n$, where $\langle i_n, j_n\rangle$ is an embedding.

The first level is a unique element domain, as the definition of D_0 indicates. The second level is formed by two elements: the undefined element $\perp_{D_1}$, and the continuous function d_1 from $\{\perp_{D_0}\}$ to $\{\perp_{D_0}\}$. The third level has four elements: one is the undefined element $\perp_{D_2}$, and the others are three continuous functions from D_1 en D_1. Recall that D_1 has two elements, $\perp_{D_1} \sqsubseteq d_1$, therefore the three functions are d_{20}, d_{21} and d_{22}, verifying $d_{20}(\perp_{D_1}) = \perp_{D_1} \sqsubseteq d_{20}(d_1) = \perp_{D_1}$, $d_{21}(\perp_{D_1}) = \perp_{D_1} \sqsubseteq d_{21}(d_1) = d_1$ and $d_{22}(\perp_{D_1}) = d_1 \sqsubseteq d_{22}(d_1) = d_1$, respectively. These three levels are represented in Figure 2.1.

There is a generalization of the embeddings that allows us to move from level k to level n throughout the injection i_{kn} and the projection j_{nk}. Notice that these functions are not inverse to each other. When we pass from a lower level to an upper one through an injection, we search for a value in the upper level whose projection corresponds to the value in the lower level. Since the upper level has more information than the lower level, there are several values that verify this condition. The most undefined value is chosen. Therefore, $i_{kn} \circ j_{nk} \sqsubseteq id_n$ and $j_{nk} \circ i_{kn} = id_k$. This property is inherited from the embeddings of consecutive levels. Figure 2.2 shows this situation for $n > k$.

Notice that $\langle D_n,\ j_n\rangle_{n\in\omega}$ is an inverse system of cpo's and $D = \lim_{\leftarrow}\langle D_n,\ j_n\rangle_{n\in\omega}$, i.e., D is the inverse limit. The initial solution is identified as $D = \{\langle x_n : n \in \omega\rangle : x_n \in D_n \wedge j_n(x_{n+1}) = x_n\}$. ψ_n denotes the projection $j_{\infty n} : D \twoheadrightarrow D_n$, and ϕ_n denotes the injection $i_{n\infty} : D_n \rightarrowtail D$. Abramsky and Ong [AO93] regard each D_n as a subset of D; i.e., if $x \in D_n$ then $\phi_n(x) = x \in D$, and if $x \in D$ then $\psi_n(x) = x_n \in D_n$. Thus $D = \bigcup_n D_n$. The denotational values are defined over the function space $D = [D \to D]_\perp$.

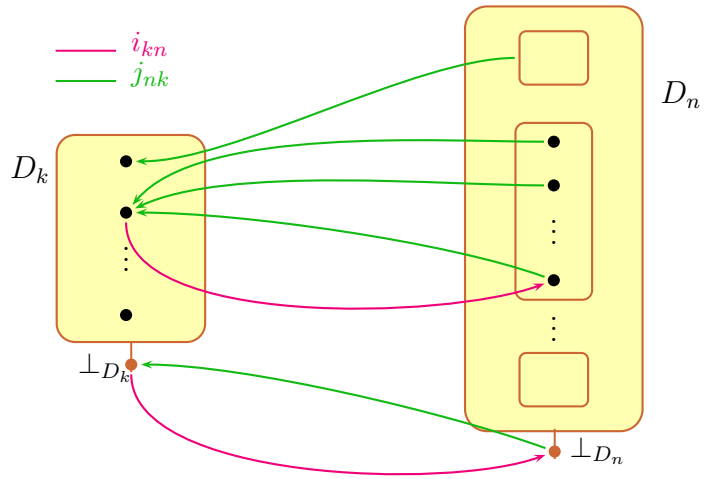


Figure 2.2: Injections and projections between levels

$$\frac{}{\lambda x.P \Downarrow \lambda x.P} \qquad \frac{M \Downarrow \lambda x.P \quad P[x := Q] \Downarrow N}{M\ Q \Downarrow N}$$

Figure 2.3: Binary relation in Λ^0

2.3.3 Applicative bisimulation

To explain the concept of applicative bisimulation defined in [Abr90], we consider a λ-calculus where the closed λ-terms, represented by Λ^0, stand for programs and the λ-abstractions for values. A binary relation $\Downarrow \subseteq \Lambda^0 \times \Lambda^0$ is defined, and its rules are shown in Figure 2.3. A term M *converges*, denoted by $M \Downarrow$, if there exists a term N such that $M \Downarrow N$, and a term M *diverges* otherwise. That is, a term either converges to a λ-abstraction, or diverges.

This definition is the basis for the *applicative bisimulation*. Abramsky and Ong in [AO93] explain that to determine the convergence of a term, one have to observe the behavior of the term at each level. Let M be a closed term. In the first level we can only see if it converges to an abstraction $\lambda x.M_1$. If the answer is affirmative, we pass N_1 as the argument of the function and we observe if the obtained expression converges, i.e. if $M_1[x := N_1]$ is convergent. We keep on performing this once and again.

A sequence of binary relations, $\langle \sqsubseteq^B_k : k \in \mathbb{N} \rangle$, are defined over Λ^0:

- $\forall M, N\ .\ M \sqsubseteq^B_0 N$.
- $M \sqsubseteq^B_{k+1} N \stackrel{\text{def}}{=} M \Downarrow \lambda x.P \Rightarrow (N \Downarrow \lambda x.Q\ \wedge\ \forall R \in \Lambda^0 . P[x := R] \sqsubseteq^B_k Q[x := R])$.
- $M \sqsubseteq^B N \stackrel{\text{def}}{=} \forall k \in \mathbb{N}\ .\ M \sqsubseteq^B_k N$.

Notice that at level 0 all the closed terms are related. We can only observe if a term reduces to a λ-abstraction, but the body of this λ-abstraction is not observable. Therefore, the only way to have some information about the body is to study how the function behaves

$$
\begin{array}{lcll}
x & \in & \mathit{Var} & \\
e & \in & \mathit{Exp} & ::= \quad x \mid \lambda x.e \mid (e\ x) \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e
\end{array}
$$

Figure 2.4: Restricted syntax of the extended λ-calculus

$$
\begin{array}{rcll}
(\lambda x.e)^* & = & \lambda x.(e^*) & \\
x^* & = & x & \\
(\texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e)^* & = & \texttt{let}\ \{x_i = (e_i^*)\}_{i=1}^{n}\ \texttt{in}\ (e^*) & \\
(e_1\ e_2)^* & = & \begin{cases} (e_1^*)\ e_2 & \text{if } e_2 \text{ is a variable} \\ \texttt{let}\ y = (e_2^*)\ \texttt{in}\ (e_1^*)\ y & \text{otherwise,} \\ & \text{where } y \text{ is a fresh variable} \end{cases} &
\end{array}
$$

Figure 2.5: Normalization of the extended λ-calculus

when an argument is applied. Two convergent terms are related in any other level, if when the same argument is passed to them, the obtained terms are related in the previous level. Finally, two terms are related if they are related in each level.

2.4 Natural semantics for lazy evaluation

The lazy natural semantics (*call-by-need*, see Section 2.1.2) presented by Launchbury in [Lau93] had a great impact in the functional paradigm. The work has been frequently cited and it has been the basis of other studies and extensions [BKT00, HO02, NH09, Ses97, vEdM07]. In Launchbury's natural semantics, expressions are evaluated in a context. The context is represented as a set of pairs (variable/expression) where all the information is shared. Moreover, the pairs are updated as expressions are substituted by their values when they are computed. This is how laziness is modeled.

Launchbury defines the semantics of an untyped λ-calculus with recursive local declarations (see Figure 2.4). A process of normalization is applied to the expressions of this calculus:

- The first step is an α-conversion, where all the bound variables (inside an abstraction or a local declaration) are renamed with fresh names. In this way, all the local variables have distinct names.

- During the second step the arguments in applications are changed to variables as it is shown in Figure 2.5. This is represented by e^*.

This normalization process simplifies the definition of the operational semantics rules since the use of distinct names makes the context of application irrelevant, and the restriction on applications prevents from introducing new closures in the semantics.

The judgements of Launchbury's natural semantics are of the form

$$\Gamma : e \Downarrow \Delta : w,$$

$$\textsc{Lam} \quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App} \quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \quad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var} \quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let} \quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : w}$$

Figure 2.6: Natural semantics

$$\begin{aligned}
[\![\lambda x.e]\!]_\rho &= F_n(\lambda\nu.[\![e]\!]_{\rho \sqcup \{x \mapsto \nu\}}) \\
[\![e\ x]\!]_\rho &= ([\![e]\!]_\rho)\downarrow_{F_n} ([\![x]\!]_\rho) \\
[\![x]\!]_\rho &= \rho(x) \\
[\![\texttt{let}\ x_1 = e_1 \ldots x_n = e_n\ \texttt{in}\ e]\!]_\rho &= [\![e]\!]_{\{\!\{x_1 \mapsto e_1 \ldots x_n \mapsto e_n\}\!\}_\rho}
\end{aligned}$$

Figure 2.7: Denotational Semantics

that is, an expression e, which is evaluated in the context heap Γ, reduces to a value w in the context heap Δ. *Heaps* are partial functions from variables to expressions. $x \mapsto e$ denotes a *binding*, i.e., a pair (variable, expression). *Values* ($w \in Val$) are expressions in *weak-head-normal-form* (whnf), i.e., they are λ-abstractions. The semantic rules are shown in Figure 2.6. During the evaluation of an expression, new bindings can be added to the heap (rule Let). Some existing bindings can be modified by updating the expressions with the calculated values (rule Var). Rule Lam indicates that values reduce to themselves without modifying the context of evaluation. In rule Var an α-conversion of the final value is needed, this is represented by $\hat{w}$. This renaming avoids name clashes and it is justify by Barendregt variable convention [Bar84]. Rule App reduces first the term e to a λ-abstraction and afterwards the application is performed by a β-reduction; the obtained expression is then evaluated. Finally, rule Let shows how local declarations are evaluated: The local declarations are added to the heap and the body of the expression is evaluated in the new context. Notice that, due to the previous normalization process, no name clashes occur when these variables are introduced in the heap.

Launchbury also gives a denotational meaning to the expressions using Abramsky's model [Abr90]. The semantic function is:

$$[\![-]\!] \ : \ \mathit{Exp} \to \mathit{Env} \to \mathit{Value}$$

where *Exp* contains the expressions of the calculus (Figure 2.4), *Value* is an appropriate domain satisfying the equation $\mathit{Value} = [\mathit{Value} \to \mathit{Value}]_\perp$ (that has been explained in Section 2.3), and *Env* is the set of evaluation environments of the free variables. The environments are functions from variables to values, that is,

$$\rho \in \mathit{Env} = \mathit{Var} \to \mathit{Value}.$$

The denotational clauses are shown in Figure 2.7, where a function that relates heaps and environments is used:

$$\{\!\{-\}\!\} \ : \ \mathit{Heap} \to \mathit{Env} \to \mathit{Env}.$$

$$\textsc{Var}\ \frac{(\Gamma, x \mapsto e) : \hat{e} \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow \Delta : w} \qquad \textsc{App}\ \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \quad (\Theta, y \mapsto x) : e' \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

Figure 2.8: Alternative natural semantics

$$\begin{aligned}
\mathcal{N}[\![e]\!]_\sigma\ \bot &= \bot \\
\mathcal{N}[\![\lambda x.e]\!]_\sigma\ (S\ k) &= F_n(\lambda\nu.\mathcal{N}[\![e]\!]_{\sigma \sqcup \{x \mapsto \nu\}}\ ) \\
\mathcal{N}[\![e\ x]\!]_\sigma\ (S\ k) &= (\mathcal{N}[\![e]\!]_\sigma\ k)\downarrow_{F_n}\ (\mathcal{N}[\![x]\!]_\sigma\ )\ k \\
\mathcal{N}[\![x]\!]_\sigma\ (S\ k) &= \sigma\ x\ k \\
\mathcal{N}[\![\texttt{let}\ x_1 = e_1 \ldots x_n = e_n\ \texttt{in}\ e]\!]_\sigma\ (S\ k) &= \mathcal{N}[\![e]\!]_{\mu\sigma'\ (\sigma \sqcup x_1 \mapsto \mathcal{N}[\![e_1]\!]_{\sigma'}\ \sqcup \cdots \sqcup x_n \mapsto \mathcal{N}[\![e_n]\!]_{\sigma'}\ )}\ k
\end{aligned}$$

Figure 2.9: Resourced denotational semantics.

This function captures the recursion generated by the local declarations. It is defined by:

$$\{\!\{x_1 \mapsto e_1 \ldots x_n \mapsto e_n\}\!\}_\rho\ =\ \mu\rho'.\rho \sqcup (x_1 \mapsto [\![e_1]\!]_{\rho'} \ldots x_n \mapsto [\![e_n]\!]_{\rho'})$$

In this definition μ represents the least fixed point operator. This function can be seen as an environment modifier. It only makes sense if environments and heaps are consistent, that is, if a variable is bound both in the environment and in the heap, then it has to be bound to values that have an upper bound.

Launchbury defines an order on environments: $\rho \leq \rho'$ if ρ' binds more variables than ρ. Furthermore, if a variable is bound in both environments then the values have to be the same. Formally, $\forall x \in \mathit{Var}\ .\ \rho(x)\ \neq \bot \Rightarrow\ \rho(x) = \rho'(x)$.

2.4.1 Properties

Launchbury establishes the *correctness* (Section 2.2.1) of the operational rules with respect to the denotational semantics. The correctness theorem proves that reductions preserve the meaning of terms, and that they only modify the meaning of heaps by adding new bindings if they are needed.

Theorem 1 (Correctness of the natural semantics.)
If $\Gamma : e \Downarrow \Delta : z$ *then for all environment* ρ, $[\![e]\!]_{\{\!\{\Gamma\}\!\}_\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}_\rho}$ *and* $\{\!\{\Gamma\}\!\}_\rho \leq \{\!\{\Delta\}\!\}_\rho$.

The given operational and denotational semantics are not as close as desired. To establish the *computational adequacy* (Section 2.2.1) Launchbury introduces two new semantics that are closer. On the one hand, he gives new operational rules for variable and application (see Figure 2.8), so than evaluation contexts are closer to denotational environments. On the other hand, he introduces a resourced denotational semantics, where its semantic function takes a new argument, the resources: one resource is consumed in each syntactic level. This mimics the derivation process for the operational semantics. Figure 2.9 shows the new denotational clauses.

Finally, Launchbury proves the computational adequacy of the alternative natural semantics with respect to the resourced denotational semantics.

$$
\begin{array}{rcl}
E & ::= & x \mid \backslash x.E \mid E_1\ E_2 \mid E_1 \# E_2 \mid \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E \\
 & \mid & \texttt{new}(y,x)E \mid x\ !\ E_1\ \texttt{par}\ E_2 \mid E_1 \bowtie E_2 \mid \Lambda[x_1\ :\ x_2].E_1 [\![E_2 \mid L \\
L & ::= & \texttt{nil} \mid [E_1\ :\ E_2]
\end{array}
$$

Figure 2.10: Jauja syntax

Theorem 2 (Computational adequacy of the alternative semantics.) *If* $\exists m \in \mathbb{N}\ .\ \mathcal{N}[\![e]\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma\ \sqcup \cdots \sqcup x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma\ )}\ (S^m\ \bot) \neq \bot$, *then there exists a heap* Δ *and a value* w *such that* $(x_1 \mapsto e_1\ \ldots\ x_n \mapsto e_n) : e \Downarrow \Delta : w$.

2.5 The language Jauja and the formal semantics of Eden

Hidalgo-Herrero defined in [Hid04] the language Jauja, a simplification of the parallel functional language Eden (introduced in Section 2.1.4) that contents its main characteristics. Jauja has two parts: a lazy λ-calculus and coordination expressions. These coordination expressions allows to introduce parallelism. Process are created explicitly and they interact with each other through communication channels. This language also incorporates non-determinism, and therefore reactivity. We use a subset of this language in this thesis.

The syntax of Jauja is shown in Figure 2.10. An expression for process creation, #, has been added to the expressions of a λ-calculus with local declarations. In order to make less restrictive the communication between processes, the construction $\texttt{new}(y,x)E$, that create *dynamic channels*, is included. The *dynamic connection*, $x\ !\ E_1\ \texttt{par}\ E_2$, allows to evaluate E_1 and E_2 in parallel and it also communicates the value of E_1 through x. The expression $E_1 \bowtie E_2$ merges two *streams* obtained from E_1 and E_2 and introduces the non-determinism. Finally, $\Lambda[x_1\ :\ x_2].E_1 [\![E_2$ allows to work with empty, nil, and non empty, $[E_1\ :\ E_2]$, lists.

2.5.1 Operational Semantics

Hidalgo-Herrero defines an operational semantics for Jauja in [Hid04]. This semantics models its main characteristics: lazy evaluation and parallelism (inside a process and between processes). This leads to a distributed model with a two-level structure. The upper level is a distributed system S formed by several parallel processes. The lower level is formed by each process represented by a heap of bindings, H_i. The two-level structure is schematized in Figure 2.11.

This structure in two levels is reflected in the operational rules defined in [Hid04]. There are two types of rules: the local rules, which express the evolution of each process; and the global rules, which show how the system changes by the creation of new processes and the communication between them.

Local rules indicate how a labelled heap evolves. A labelled heap is a collection of labelled bindings: A stands for active bindings, B for blocked ones (waiting for the evaluation of other bindings), and I for inactive bindings (either already evaluated, or not demanded yet). Each rule focuses on an active binding. For instance, rule **(app-demand)**

$$H\ +\ \{x \overset{I}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\ y\ \longrightarrow\ H\ +\ \{x \overset{A}{\mapsto} E, \theta \overset{B}{\mapsto} x\ y\}$$

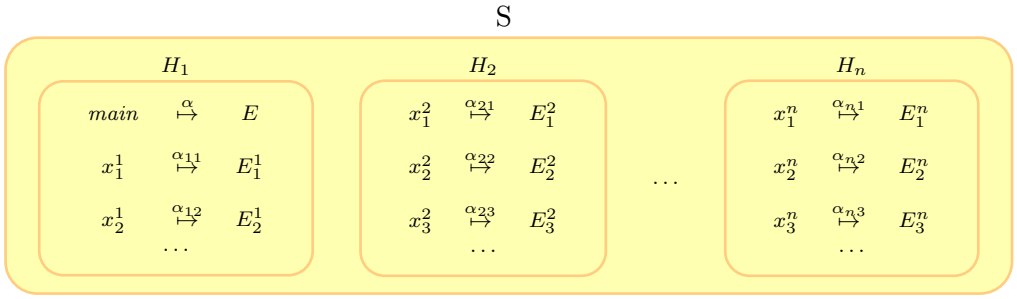


Figure 2.11: Distributed model

indicates that in order to evaluate an application, the binding that refers to the body is activated, while the demanded binding is blocked.

The evolution of the whole system is performed by the transition relation

$$\Longrightarrow = \overset{par}{\Longrightarrow} ; \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{Unbl}{\Longrightarrow} .$$

The first step is $\overset{par}{\Longrightarrow}$ that manages the execution of the active bindings in the system. The semantics ranges between a minimum and a maximum, depending on how much parallel work is achieved. The minimum semantics corresponds to the situation where there is no speculative work and only the bindings demanded by the main variable are allowed to evolve. By contrast, in the maximum semantics all the active bindings evolve. Next, the rule $\overset{comm}{\Longrightarrow}$ is applied, and all the possible communications are accomplished. The rule $\overset{pc}{\Longrightarrow}$ indicates that all the possible process creations are fulfilled. Finally, the rule $\overset{Unbl}{\Longrightarrow}$ reorganizes the labels of the bindings: blocked bindings that were pending of a value that has been already obtained are unblocked; those process creations that could not be completed are blocked; and the bindings that are needed to achieve pending process creations and communications are demanded.

2.5.2 Denotational semantics

The denotational semantics of `Jauja` is not used in this thesis. It is a continuation semantics which models laziness and the possible lateral effects of the evaluation of an expression. This semantics reflects not only the denotational value of an expression, but also the parallelism of the language. For instance, the denotation of $x_1 \# x_2$ shows the value of the functional application and also the corresponding process creation and communications. To formalize this semantics, the definition of several semantic domains is required. The evaluation function type is:

$$\varepsilon :: \texttt{Exp} \rightarrow \texttt{IdProc} \rightarrow \texttt{ECont} \rightarrow \texttt{Cont},$$

where one has to provide the expression to be evaluated, `Exp`, the process where the evaluation is performed, `IdProc`, and the expression continuation with the information of what has to be done with the obtained value, `ECont`. The function returns a continuation, `Cont`, that accumulates the lateral effects of the evaluation and those of the expression continuation.

2.6 λ-calculus representations

Pitts explains in [Pit13] that when a programming language is defined, a concrete syntax to generate correct terms (chains of symbols) is specified. But some details of this syntax are irrelevant for the meaning of the programs.

This section focuses on the α-conversion problem generated by the syntax of the λ-calculus. One of the main problems is due to the capture of free names when a substitution is accomplished. For this reason, α-equivalent terms are usually considered, i.e., terms that only differ on the name of bound variables. If the chosen names cause problems (clash of names) when building a formal proof, then another α-equivalent term, whose names do not clash with the free names, is considered. This procedure is known as Barendregt variable convention [Bar84].

However, the application of Barendregt variable convention cannot be always performed [UBN07]. When building a proof by induction, some steps can be proved for variables that are fresh enough but not for any name.

There are several alternatives to the use of a named notation. Some of them are explained below.

2.6.1 The de Bruijn notation

To give a formalization of the λ-calculus compatible with computers, de Bruijn proposed in [dB72] a *namefree* notation, where names are substituted by numbers. Although the purpose of this notation was not to deal with the problems explained above, the namefree notation turns out to solve them. In order to explain the ideas of the author, we consider a λ-caculus with variables, abstractions and applications, without local declarations and constants, i.e., $t ::= x \mid \lambda x.t \mid a(t,t)$. When using this notation all α-equivalent terms are equal. The following example explains how to achieve the unique representation of a term.

Example 1 *Consider the expression*

$$\lambda x.\lambda y.a(\lambda z.a(a(w,z),t),y)$$

To translate this term into the namefree notation, a list of names with the free variables of the term is needed. In this case there are two free variables, w and t. For instance, choose the list $[w,t]$. We consider now the syntactic tree of the expression and we complete it with some nodes on top: λw and λt. We associate a number to each name. This number is the reference depth*, and it indicates the number of λ's that has to be passed in the syntactic tree until arriving to the corresponding λ. Figure 2.12 shows this construction. The nodes that refer to the free variables are highlighted in a different color.*

To finish we replace the names of variables with the obtained numbers. In our example the term with the namefree notation is $\lambda.\lambda.a(\lambda.a(a(5,1),4),1)$. □

As de Bruijn observed, the namefree notation has a huge disadvantage. Although it is useful to work with computers, it is not intuitive, and it is hard to work with it. For example, when executing an application, a λ disappears from the syntactic tree and the indices have to be recalculated. This is easy for a machine, that has only to apply some rules, but when a human being is working with abstract terms, these changes deeply complicate the syntax of the expression.

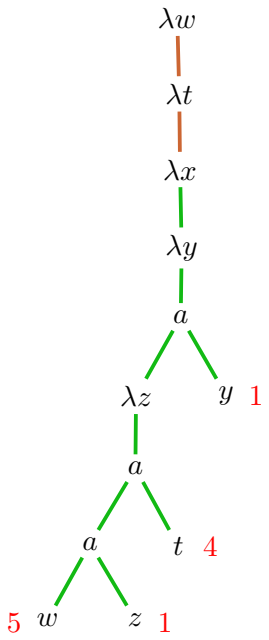


Figure 2.12: A de Bruijn example

$$t \;:=\; \texttt{bvar}\ i \;\mid\; \texttt{fvar}\ x \;\mid\; \texttt{abs}\ t \;\mid\; \texttt{app}\ t\ t$$

Figure 2.13: λ-calculus, locally nameless representation

2.6.2 Locally nameless representation

To deal with the problems derived from the α-conversion, we choose the *locally nameless representation*. This notation was introduced by de Bruijn [dB72] as an alternative to the namefree notation explained in Section 2.6.1. The locally nameless representation uses indices for the bound variables but retain the names of the free variables. This notation has been used in several works [Gor94, Ler07, ACP+08], but Charguéraud [Cha11] has developed a complete description of this representation. He shows the syntax of a λ-calculus with this notation (see Figure 2.13) and he gives some operations that are needed to work with these terms.

The main operations on locally-named-terms are the *variable opening* and *variable closing*. The former allows to study the body of an abstraction $\texttt{abs}\ t$. When the term t is opened with a fresh name x, represented as t^x, the term is modified and the variables ($\texttt{bvar}\ i$) bound to the abstraction ($\texttt{abs}\ t$) become free ($\texttt{fvar}\ x$). Next example shows this situation:

Example 2 *Consider the term* $t \equiv \texttt{abs}\ u$ *where*

$$u \equiv (\texttt{app}\ (\texttt{abs}\ (\texttt{app}\ (\texttt{bvar}\ 1)\ (\texttt{bvar}\ 0)))\ (\texttt{bvar}\ 0)).$$

There are two variables in u, that refers to the abstraction. When opening the body with variable x the following term is obtained:

$$u^x \equiv \texttt{app (abs (app (fvar } x\texttt{) (bvar 0))) (fvar } x\texttt{)}.$$

□

Variable closing is the inverse of variable opening under some freshness conditions. Variable x of a given term becomes bound when building an abstraction with this term.

Example 3 *Consider the term*

$$u \equiv \texttt{app (abs (app (fvar } x\texttt{) (bvar 0))) (fvar } x\texttt{)}.$$

Variable closing is used to build an abstraction binding variable x:

$$\texttt{abs } ({}^{\backslash x}u) \equiv \texttt{abs (app (abs (app (bvar 1) (bvar 0))) (bvar 0))}.$$

□

This locally nameless representation allows to build terms without correspondence to named λ-terms. The predicate *locally closed* is defined to identify well-formed terms. We also find detailed definitions of substitution and free variables of a term in [Cha11].

Charguéraud uses cofinite quantification in some of the rules that define the mentioned predicates and functions. This cofinite quantification was previously studied by Charguéraud et al. in [ACP+08]. Cofinite quantification is in-between existential and universal quantification. When proving by induction we may need to rename a variable that has been used to open an abstraction, so that clash of names is avoided. Cofinite quantification solves this problem, since it establishes that the hypothesis is verified for any variable but a finite quantity of them. We have worked in this thesis with cofinite quantification to express some semantic rules in their locally nameless version.

2.7 Proof assistants

Different tools to work with mathematical proofs have been developed during the last years. Geouvers summarizes the history of proof assistants in [Geu09]. Automated theorem provers are different from proof assistants. The former are systems with some procedures that allow to automatically prove some formulation, while the latter need to "be guided" by a human being in the tricky points of a proof. The user applies some tactics to guide the machine to build the proof.

There are many proof assistants nowadays with different characteristics. The most popular are Isabelle [isa14], Agda [agd14], PVS [pvs14] and Coq [coq14]. Next table outlines some of the main characteristics of each one [Wie06]:

Name	**Higher-order logic**	**Dependent types**	**Small kernel**	**Proof automation**	**Proof by reflection**	**Code generation**
Isabelle	Yes	No	Yes	Yes	Yes	Yes
Agda	Yes	Yes	Yes	No	Yes	Yes
PVS	Yes	Yes	No	Yes	No	Yes
Coq	Yes	Yes	Yes	Yes	Yes	Yes

Only the rules that form the kernel of a proof assistant must be proven to be correct; the others are built from them. For this reason it is important to have a small kernel.

We have used COQ in thesis to extend some definitions and results implemented by Charguéraud in [Cha11]. They refer to the locally nameless representation given in Section 2.6.2.

Chapter 3

What have we got?

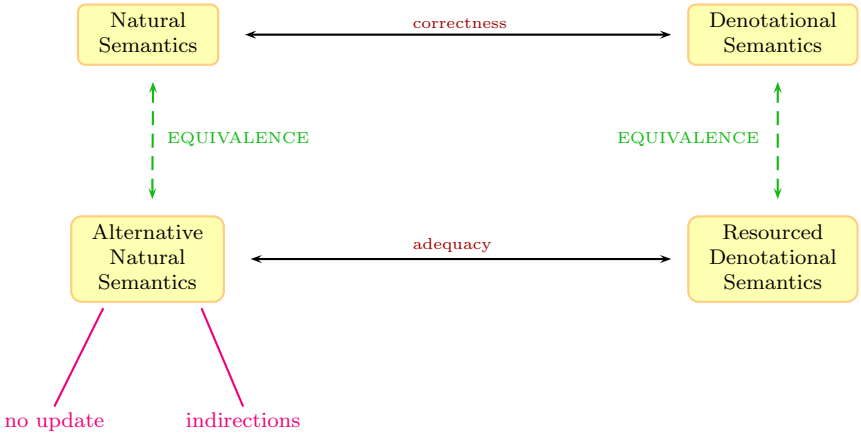
In this chapter we gather and discuss the main contributions of this thesis. We do not follow a historical line. Instead we enclose the concepts by themes.

As it was indicated in the Introduction (Chapter 1), our results can be classified into two groups: the research done to prove Launchbury's computational adequacy (Section 2.4) and the extension of some results to a distributed model.

3.1 Computational adequacy

First, we briefly explain the problems concerning the proof of Launchbury's computational adequacy [Lau93]. After that, we focus on how we have solved some of these problems. In the chapter devoted to the future work, we also explain how we are working to solve the rest of the problems.

The following scheme shows the semantics defined by Launchbury (given in Section 2.4) and the relation between them:



To prove the equivalence between the natural semantics and a standard denotational semantics, Launchbury concentrated on the proofs of the correctness and computational adequacy (Section 2.4.1). As we explained in Section 2.2.1, correctness implies that the meaning of a term does not change during its evaluation; computational adequacy determines that an expression reduces to a value in the operational semantics if and only if its denotational value is defined. To prove the computational adequacy, Launchbury introduced two new semantics: an alternative natural semantics and a resourced denotational semantics. As we mentioned in Section 2.4, the former is a natural semantics without update of closures and where applications are accomplished via indirections instead of

carrying out β-reductions. The latter is a denotational semantics where terms are undefined when the number of resources provided are insufficient. Launchbury proved the computational adequacy between the new versions of the semantics, but he only hinted how to prove the equivalence with the original semantics. These indications turn out to be insufficient.

These problems are detailed in the following sections. Firstly, we explain how we have proved the equivalence between the two denotational semantics. Secondly, we describe how we have dealt with the equivalence of the two operational semantics.

3.1.1 Function space with resources (Publication P1)

In the denotational semantics with resources given by Launchbury, the denotation of a term can be undefined because of two reasons: either because the corresponding value is $\bot$, or because there are not enough resources to evaluate the term. Launchbury asserted that the standard denotational semantics and its resourced version compute the same values when infinite resources are provided. However, the domains for each semantics are different and, therefore, it can not be an equality. Instead of this, we need to find out a way to relate them.

To represent the values of the resourced semantics, we consider the domain equation $E = [[C \to E] \to [C \to E]]_\bot$, where C stands for the resources, instead of the usual function space equation domain $D = [D \to D]_\bot$ shown in Section 2.3. In the new domain E, the depth of application allowed for evaluation is bounded. We follow Abramsky's steps in the construction of D (Section 2.3.2) to build up E by considering C as the initial solution of the equation $C = C_\bot$. The elements of C are represented by $\bot$, $S(\bot)$, $S^2(\bot)$,... where S is the successor function. The *finite aproximations of* E are defined as follows:

$$\begin{array}{lcl} E_0 & \stackrel{\text{def}}{=} & \{\bot_{E_0}\}, \text{ and} \\ E_{n+1} & \stackrel{\text{def}}{=} & [[C \to E_n] \to [C \to E_n]]_\bot. \end{array}$$

Each level has a greater definition capacity than the previous one. What is more, each level is contained in the next one:

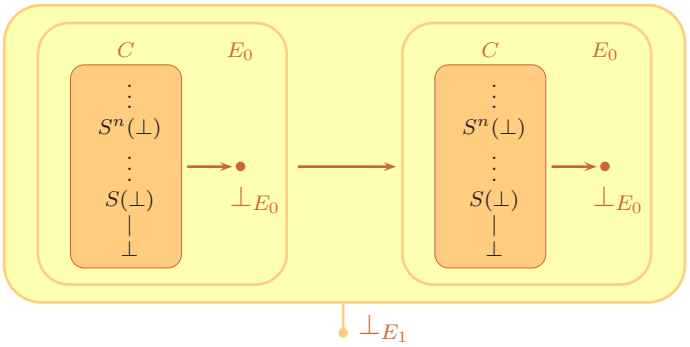
$$\begin{array}{l} E = [[C \to E] \to [C \to E]]_\bot \\ \quad | \\ \quad \vdots \\ \quad | \\ E_{n+1} = [[C \to E_n] \to [C \to E_n]]_\bot \\ \quad | \\ \quad \vdots \\ \quad | \\ E_1 = [[C \to E_0] \to [C \to E_0]]_\bot \\ \quad | \\ E_0 = \{\bot_{E_0}\} \end{array}$$

Next we show graphically the first levels of the construction of E. E_0 has a unique element, the undefined element:

$$E_0 = \{\bot_{E_0}\}$$

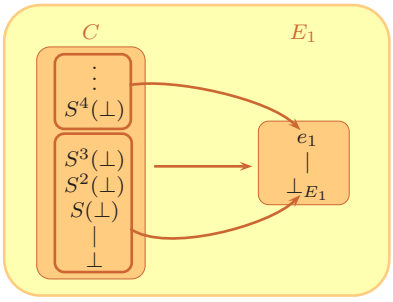
The construction of E_1 corresponds to:

$$E_1 = [[C \to E_0] \to [C \to E_0]]_\perp$$



The elements of E_1 are either the undefined element, represented as $\perp_{E_1}$, or a function from $C \to E_0$ to $C \to E_0$. Let $A_0 = C \to E_0$. There is a unique function in A_0, named a_0, that returns the undefined value of E_0 no matter how many resources are provided. Thus, E_1 has only two elements: the undefined element and the function $e_1 : a_0 \mapsto a_0$.

To construct $E_2 = [[C \to E_1] \to [C \to E_1]]_\perp$ we consider $A_1 = [C \to E_1]$. There are infinitely many functions in A_1 that return e_1 if enough resources are provided, and the undefined value of E_1 otherwise. For instance, the function $a_{1,4}$ returns the undefined value if less than four resources are provided, and e_1 if there are at least four resources. The following figure shows this example:



We add to E_2 one more function, $a_{1,\infty}$, that always returns the undefined value of E_1 independently of the number of resources.

Consequently, the elements of E_2 are: the undefined value of E_2, represented as $\perp_{E_2}$, and all the continuous functions from A_1 to A_1 verifying that if $a_{1,m}$ is more defined than $a_{1,n}$, then the image of $a_{1,m}$ is also more defined than the image of $a_{1,n}$.

Once the domain E has been built up, our next step is to relate its functions to those in D when infinite resources are provided. We use the concept of *applicative bisimulation* defined by Abramsky (Section 2.3.3). This establishes that two functions are "similar" if they produce "similar values" when applied to "similar arguments", that is, if they behave "identically" in their respective domains. First, we define recursively the *similarity* of functions at each level (represented as $\lhd\rhd_n$, for $n \geq 0$): the undefined values of D_{n+1} and E_{n+1} are *similar*, and two functions $d \in D_{n+1}$ and $e \in E_{n+1}$ are *similar* if, whenever their arguments are *similar* at level n, they produce *similar* values at level n. Figure 3.1 shows this idea.

The final similarity relation between domains D and E, $\lhd\rhd$, is defined as the least relation that verifies that two values of D and E are related if their projections are similar at each level.

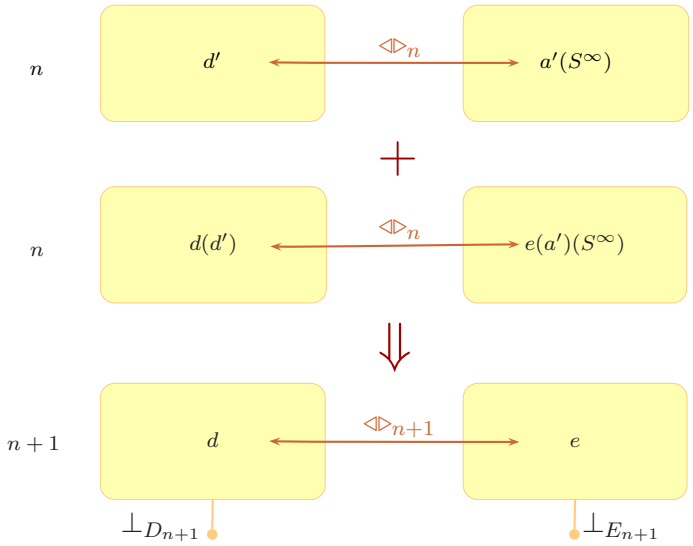


Figure 3.1: Intuition of similarity

There is an alternative characterization of the relation $\lhd\rhd$ that asserts that two values of D and E are related either if both are undefined, or if similar values are obtained when applied to similar arguments. This statement is formally written in the next proposition:

Proposition 1
Let $d \in D$, $e \in E$. $d \lhd\rhd e$ if and only if either:

- $(d = \bot_D \wedge e = \bot_E)$, *or*
- $(d \neq \bot_D \wedge e \neq \bot_E) \wedge \forall d' \in D. \forall a' \in [C \to E]. d' \lhd\rhd a'(S^\infty) \Rightarrow d(d') \lhd\rhd e(a')(S^\infty)$.

Finally we apply this result to prove the equivalence of the two denotational semantics proposed by Launchbury. For this purpose we extend the concept of *similarity* to environments. An environment ρ of the standard denotational semantics is *similar* to an environment σ of the resourced denotational semantics when infinite resources are provided, if the values associated to each variable in their respective domains are similar. We can prove now the following theorem:

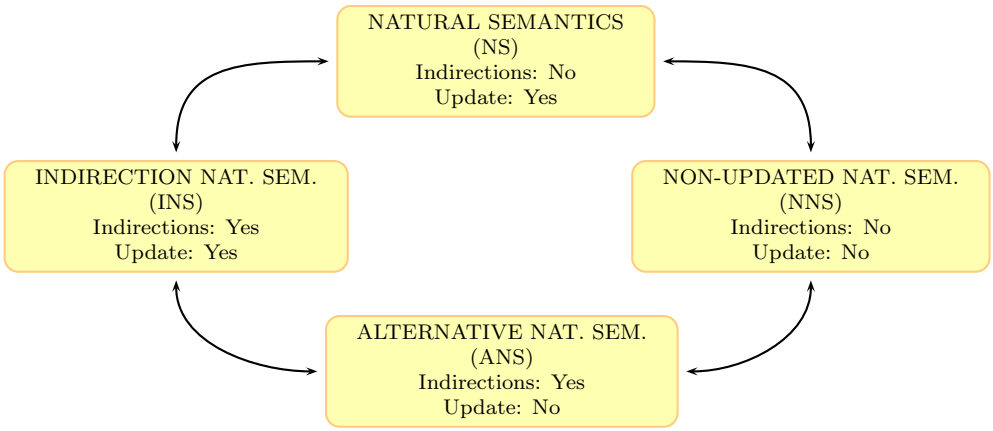
Theorem 3 (Equivalence of denotational semantics.)
If $e \in Exp$ and $\rho \lhd\rhd \sigma$, then $[\![e]\!]_\rho \lhd\rhd \mathcal{N}[\![e]\!]_\sigma\ (S^\infty)$.

Summary of results.

1. Construction of the initial solution of the domain equation $E = [[C \to E] \to [C \to E]]_\bot$, where C stands for the domain of resources.
2. Definition of the relation of *similarity* between the values of the domain E and the values of the standard function space $D = [D \to D]_\bot$.
3. Application of the previous result to prove the equivalence between the denotational standard semantics and the resourced denotational semantics for the lazy λ-calculus.

3.1.2 Alternative natural semantics

The changes introduced by Launchbury in the alternative semantics (see Section 2.4) have more consequences than expected. When an expression is evaluated in the alternative semantics, the final heap is bigger than the one obtained by applying the rules of the original natural semantics, i.e., it includes more bindings. Moreover, the closures are not updated with the computed values. In order to study these changes separately, we define two intermediate semantics, as it is shown in the following figure:



The *natural semantics with indirections* (INS) keeps the updating of bindings but it introduces indirections when applications are evaluated. The *natural semantics with non-update* (NNS) does not update the bindings in the heap and does not introduce indirections. Later on, we observe the differences between the final heaps and values obtained with each semantics.

The modifications introduced by the alternative rules are only a part of the problem when trying to establish the equivalence between the semantics. Working with a named representation implies dealing with α-equivalence. To avoid this complication we use the *locally nameless* representation explained in Section 2.6.2. Moreover, this representation helps the formalization of results using proof assistants.

Locally nameless representation (Publication P2 and TechRep R1)

We extend with recursive local declarations the locally nameless representation of the λ-calculus given by Charguéraud [Cha11]. Since bound variables have no names, we avoid working with equivalence classes and chosing a representative of the class. Using this notation, there is a unique representation for all the closed terms with the same semantical meaning.

As we have seen in Section 2.6.2, Charguéraud has developed a locally nameless representation for a language with variables, abstractions and applications. Following the indications of Charguéraud, in the presence of local declarations we represent bound variables with two indices: the first index determines to which constructor –either an abstraction or local declaration– the variable is bound (see the explanation in Section 2.6.1 and Example 1); the second index reveals the expression to which we refer inside the constructor (if the constructor is a local declaration, this second index indicates which of the local variables is concerned). The new syntax is given in Figure 3.2, where *Var* represents the set of *variables* that can be either bound (`bvar` i j) or free (`fvar` x). The following example illustrates how the indices for the bound variables are determined.

$$
\begin{array}{lcl}
x \in Id & & i, j \in \mathbb{N} \\
v \in Var & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in LNExp & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

Figure 3.2: Locally nameless syntax

Example 4 *The expression e is a λ-abstraction whose body is a* let*-expression with two local declarations.*

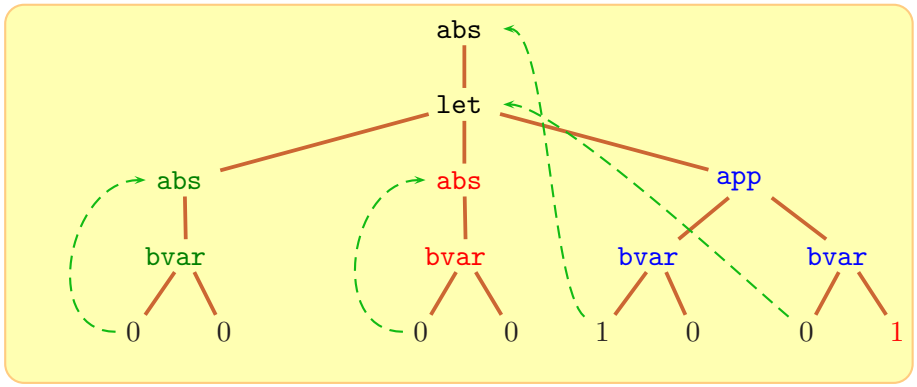
$$
\begin{aligned}
e \equiv \lambda z.\texttt{let}\ & x_1 = \lambda y_1.y_1, \\
& x_2 = \lambda y_2.y_2, \\
\texttt{in}\ & (z\ x_2)
\end{aligned}
$$

This expression is represented in the locally nameless notation by t:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

The expression bounded to x_1 appears in green, the one bounded to x_2 in red, and the main term in blue.

Let us see the syntactic tree to clarify how the indices point to the constructors:



Notice that the second index of the variable that points to the let *expression is red. This indicates that it refers to the local declaration that appears with that color inside the* let *constructor.* □

Although Charguéraud suggests how the language should be extended with local declarations, this study is not fully developed. The opening of a term and the rules for local closure are explained, but the rest of operators and functions over terms are not defined in [Cha11]. We have extended the following functions: The closure of terms, the predicate of local closure at level k, the free variables of a term, the substitution and the concept of freshness. We have also extended some results such as the one which asserts that the opening and closing of a term are inverse functions under some conditions.

Once we have established the operators and functions needed to work with the locally nameless representation, we have reformulated Launchbury's semantic rules (Figure 2.4 in Section 2.4). First we have redefined the heaps as sets of pairs of the form $(\texttt{fvar}\ x, t)$, and we have readjusted all the concepts related with them: domain of a heap, names that appear in a heap, well-formed heaps and substitutions on heaps. Moreover, we have proved some useful properties derived from this notation.

$$\text{LNLet}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \quad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Figure 3.3: Variable local declaration (locally nameless representation)

We have already explained in Section 2.6.2 that the locally nameless representation allows the formation of incorrect terms, in the sense that they do not correspond to any term in the λ-calculus. Therefore, we have added some premises to the semantic rules to ensure that they are restricted to well-formed heaps and correct terms, i.e., *locally closed* terms.

Here we just highlight the new version of the rule for local declarations, that is, the LNLet rule shown in Figure 3.3. Tuples are represented as $\overline{t}$, if they refer to terms, or $\overline{x}$, if they refer to variables. To evaluate the expression $\texttt{let}\ \overline{t}\ \texttt{in}\ t$, the local declarations $\overline{t}$ are added to the heap. Since we are using the locally nameless representation, the names of the local variables are not explicit and we need to choose a list of fresh names $\overline{x}$. These names are used to open the local terms and the body of the expression. The evaluation of $t^{\overline{x}}$ produces a final heap and value that are dependent on the chosen names. In the final heap these names, $\overline{x}$, are explicitly shown, and the rest of names are represented by $\overline{z}$. We use cofinite quantification in the rule LNLet. As it is explained in [Cha11], the advantage of this quantification is that the side-conditions concerning the freshness of $\overline{x}$ are not detailed. These conditions are implicit in the set L. This set contains all the names that must be avoided during the reduction, in particular those that appear in other parts of the derivation. Our cofinite rule is different from the ones described in [Cha11], because the chosen names appear also in the conclusion. As a result, we have that the new names $\overline{x}$ can be replaced by any list of names $\overline{y}$ that are not in L.

After translating the semantic rules of the natural semantics and its alternative version into the locally nameless notation[1], we have proved some properties, for instance:

Regularity ensures that the judgements produced by these reduction systems yield only well-formed heaps and locally closed terms;

Renaming indicates that the evaluation of a term is independent of the fresh names chosen during the derivation;

Introduction establishes a correct existential rule for each cofinite rule.

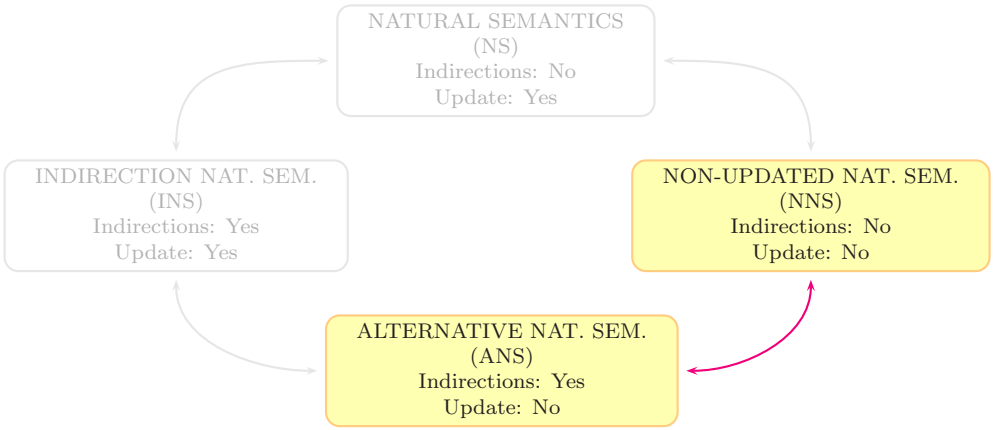
Summary of results.

1. Locally nameless representation of the λ-calculus extended with recursive local declarations.
2. Locally nameless representation of Launchbury's natural semantic rules and its alternative version.
3. Properties of the reduction systems (regularity, introduction and renaming).

[1]The translation of the alternative rules into the locally nameless notation do not appear in P2 but in P3, which is explained in next section. However, we have considered more convenient to refer them in this section that is devoted to the locally nameless representation.

An Indirection relation (Publication P3 and TechRep R2)

Once the semantic rules have been translated into the locally nameless representation, we can explore the equivalence between the non-updated semantics (NNS) and the alternative semantics (ANS). We want to prove that the result obtained by both semantics when evaluating a term in a given context are the "same". This equivalence is highlighted in the following diagram:



Both semantics use the same semantic rules except for the application rule. The NNS uses the LNApp rule, where the application is performed by a β-reduction, while the ANS applies the ALNApp rule, where an *indirection* is introduced in the heap. Although this distinction seems to be irrelevant, the final heaps obtained in derivations are different. Those obtained by evaluating with the ANS are bigger than the ones obtained by using the rules of the NNS. Notice that an "extra" binding is introduced in the heap each time an application is performed. This binding is an indirection, i.e., a name bound to a name. Therefore, final heaps cannot be the same and we study the relation between them. Our first idea was just to remove the indirections and check for the equality of the resulting heaps. However, this is insufficient, since some closures in the final heap could depend on the removed bindings.

Next example shows this problem. $\Downarrow^N$ indicates a derivation for NNS is being applied, while $\Downarrow^A$ refers to ANS:

Example 5 *Consider the following term:*

$$t \equiv \texttt{let abs (bvar 0 0) in app (abs } s\texttt{) (bvar 0 0)}$$
$$s \equiv \texttt{let abs (bvar 0 0), app (bvar 0 0) (bvar 1 0) in abs (bvar 0 0)}$$

If we evaluate t (in the context of an empty heap) we obtain abs (bvar 0 0) *as final value in both cases, but different final* heaps *are built:*

$$\{\,\} : t \Downarrow^N$$

$x_0 \mapsto$ abs (bvar 0 0),
$x_1 \mapsto$ abs (bvar 0 0),
$x_2 \mapsto$ app (fvar x_1) (fvar x_0)

$$\{\,\} : t \Downarrow^A$$

$x_0 \mapsto$ abs (bvar 0 0),
$x_1 \mapsto$ abs (bvar 0 0),
$x_2 \mapsto$ app (fvar x_1) (fvar y)
$y \mapsto$ (fvar x_0)

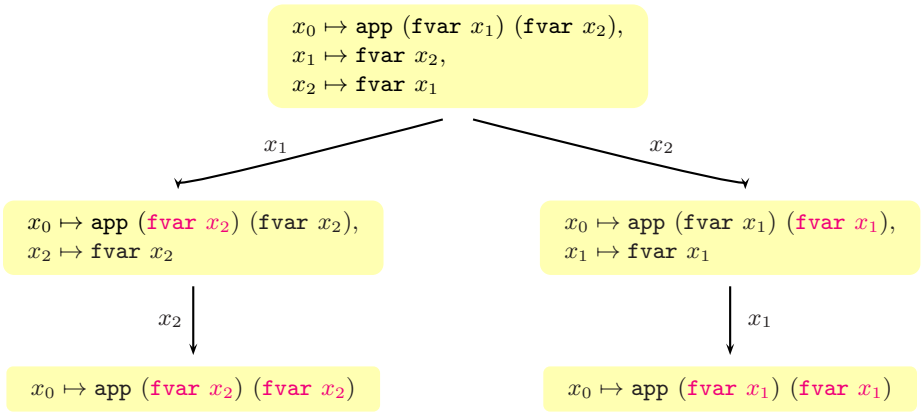
We observe that the final heap obtained by evaluating with the alternative semantics has an extra binding; moreover, the term associated to the name x_2 depends on it. □

This example shows that we need to replace all the occurrences of the name that is going to be eliminated for the name to which it is bound.

We thought to find a method to detect which indirections come from the application of the ALNApp rule, but we have decided to deal with a more general setting. We have defined a relation between heaps based on indirections, without caring about their origin. Two heaps are related if the smallest one is obtained by removing some indirections from the biggest one (to remove a binding implies also the substitution of names).

However, the order in which indirections are deleted is significant in the case of crossed references, as the following example shows:

Example 6 *The heap* $\Gamma = \{x_0 \mapsto \texttt{app}\ (\texttt{fvar}\ x_1)\ (\texttt{fvar}\ x_2), x_1 \mapsto \texttt{fvar}\ x_2, x_2 \mapsto \texttt{fvar}\ x_1\}$, *has two indirections. We will see that the order in which they are removed does affect the result:*



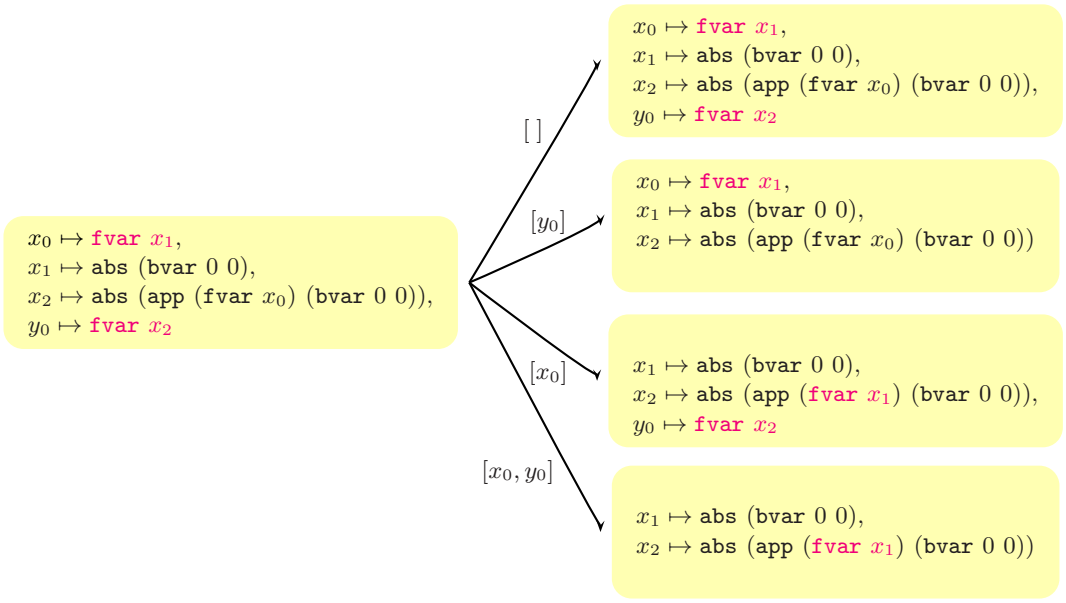
□

Notice that in the previous example both names, x_1 and x_2, are undefined in the final heap. Thus, for our relation on heaps we ignore undefined variables. Two heaps are indirection related when the smallest one is obtained from the biggest one up to undefined variables by deleting some indirections. Undefined variables are those that do not belong to the heap domain, but may appear on the right-hand-side terms. To formalize this, we first define the *equivalence of terms in a given context*, then this relation is used to define the *equivalence of heaps in a given context*. Finally, the *indirection relation* ($\succsim_I$) on heaps is based on the latter equivalence.

A given heap can be related to several heaps, as the following example illustrates:

Example 7 *We show all the heaps indirection related to*

$$\Gamma = \{\ x_0 \mapsto \texttt{fvar}\ x_1, x_1 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{bvar}\ 0\ 0)),\ y_0 \mapsto \texttt{fvar}\ x_2\ \}$$



□

We extend this relation to (heap, term) pairs, and enunciate and prove the main result of this part. The following theorem establishes that if a term evaluates in a given context with the NNS, it also evaluates with the ANS and vice versa. What is more, the final (heap, term) pairs are indirection related.

Theorem 4 (Equivalence ANS and NNS.)

EQ_AN $\quad \Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in \mathit{LNHeap} \,.\, \exists w_N \in \mathit{LNVal} \,.\, \Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \precsim_I (\Delta_N : w_N)$

EQ_NA $\quad \Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in \mathit{LNHeap} \,.\, \exists w_A \in \mathit{LNVal} \,.\, \exists \overline{x} \subseteq \mathtt{dom}(\Delta_N) - \mathtt{dom}(\Gamma) \,.\, \exists \overline{y} \subseteq \mathit{Id} \,.\, |\overline{x}| = |\overline{y}| \wedge$
$\Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \precsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$

The second part of the theorem, EQ_NA indicates that a renaming may be needed. This renaming only affects the names that have been added during the derivation. A name occurring in a term can disappear during the evaluation with the NNS. This could lead to the introduction of this name in the heap as "fresh". But this is impossible when evaluating with the ANS. Next example shows this situation:

Example 8 *Consider the term*

$$t \equiv \mathtt{let\ abs\ (bvar\ 1\ 1), let\ abs\ (bvar\ 0\ 0)\ in\ (bvar\ 0\ 0)\ in\ app\ (bvar\ 0\ 0)\ (fvar}\ z)$$

During the evaluation of t with the NNS in the empty context, the name z disappears from the term and the heap, while when applying the ANS rules it is introduced in the heap via an indirection.

We first show some parts of the derivation with the NNS:

$$
\begin{array}{l}
\{\,\} : t \\
\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{fvar}\ z) \\
\\
\qquad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{fvar}\ x_0 \\
\qquad\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1 \\
\qquad\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1 \\
\qquad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1 \\
\\
\qquad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{fvar}\ x_1 \\
\qquad\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0) \\
\qquad\quad \vdots \\
\qquad\quad \left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \\
\qquad \left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \\
\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \\
\{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
\end{array}
$$

When the β-reduction is performed, the name z is neither in the term nor in the heap. Hence, when the fresh name x_2 is introduced in the heap, we could have chosen z.

By contrast, when evaluating with the alternative semantics, z is introduced in the heap and, consequently, can not be chosen as a fresh name:

$$
\begin{array}{l}
\{\,\} : t \\
\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{fvar}\ z) \\
\\
\qquad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{fvar}\ x_0 \\
\qquad\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1 \\
\qquad\quad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1 \\
\qquad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs fvar}\ x_1 \\
\\
\qquad \{x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), y \mapsto \texttt{fvar}\ z\} : \texttt{fvar}\ x_1 \\
\qquad\quad \left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), y \mapsto \texttt{fvar}\ z, \\ x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0) \\
\qquad\quad \vdots \\
\qquad\quad \left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \\
\qquad \left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \\
\quad \left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \\
\left\{ \begin{array}{l} x_0 \mapsto \texttt{abs}\ (\texttt{fvar}\ x_1), x_1 \mapsto \texttt{let abs}\ (\texttt{bvar}\ 0\ 0)\ \texttt{in}\ (\texttt{bvar}\ 0\ 0), \\ y \mapsto \texttt{fvar}\ z, x_2 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0) \end{array} \right\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)
\end{array}
$$

□

The proof of Theorem 4 cannot be achieved directly by rule induction. In the subderivations we no longer have the same term to be evaluated in the same context, but heaps and terms that are indirection related. For this reason we have proved a more general result (Proposition 2) that establishes the relation between the two semantics when indirection related (heap, term) pairs are evaluated. Notice that if there exists a derivation for a term, then there exists infinitely many derivations depending on the chosen fresh names. The proposition establishes that given two indirection related (heap, term) pairs, if there exists a derivation with the ANS for the biggest heap, then there exists a corresponding derivation with the NNS for the smallest heap, and vice versa.

$$
\begin{array}{rcl}
x,\ y & \in & \mathit{Var} \\
E & \in & \mathit{EExp} \\
E & ::= & x \mid \lambda x.E \mid x\ y \mid x\#y \mid \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x
\end{array}
$$

Figure 3.4: Syntax

Proposition 2

EQ_IR_AN
$$
\begin{array}{l}
(\Gamma_A : t_A) \gtrsim_I (\Gamma_N : t_N) \wedge \forall \overline{x} \notin L \subseteq \mathit{Id}\,.\,\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}} \\
\wedge\ {}^{\backslash\overline{x}}(\overline{s}_A{}^{\overline{x}}) = \overline{s}_A \wedge {}^{\backslash\overline{x}}(w_A{}^{\overline{x}}) = w_A \\
\Rightarrow \exists \overline{y} \notin L\,.\,\exists \overline{s}_N \subset \mathit{LNExp}\,.\,\exists w_N \in \mathit{LNVal}\,. \\
\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}} \wedge {}^{\backslash\overline{z}}(\overline{s}_N{}^{\overline{z}}) = \overline{s}_N \wedge {}^{\backslash\overline{z}}(w_N{}^{\overline{z}}) = w_N \wedge \overline{z} \subseteq \overline{y} \\
\wedge\ ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})
\end{array}
$$

EQ_IR_NA
$$
\begin{array}{l}
(\Gamma_A : t_A) \gtrsim_I (\Gamma_N : t_N) \wedge \forall \overline{x} \notin L \subseteq \mathit{Id}\,.\,\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}} \\
\wedge\ {}^{\backslash\overline{x}}(\overline{s}_N{}^{\overline{x}}) = \overline{s}_N \wedge {}^{\backslash\overline{x}}(w_N{}^{\overline{x}}) = w_N \\
\Rightarrow \exists \overline{z} \notin L\,.\,\exists \overline{s}_A \subset \mathit{LNExp}\,.\,\exists w_A \in \mathit{LNVal}\,. \\
\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}} \wedge {}^{\backslash\overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \wedge {}^{\backslash\overline{y}}(w_A{}^{\overline{y}}) = w_A \wedge \overline{z} \subseteq \overline{y} \\
\wedge\ ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})
\end{array}
$$

The proof of this proposition requires of several technical lemmas where we show how the indirection relation is conveyed to the subderivations. These auxiliary lemmas are used to prove the inductive cases. For example, if two (heap, term) pairs are indirection related and the terms are applications, then the pairs formed by the heaps and the bodies of the applications are also indirection related. In the case of local declarations, the heaps extended with these local declarations and the main terms are also related.

Summary of results.

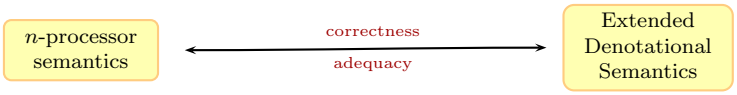
1. Equivalence relation between heaps that define the same free variables, but whose closures may differ in the undefined free variables.
2. Preorder that relates two heaps when the former is transformed into the second by the elimination of indirections ($\gtrsim_I$).
3. Extension of the preorder to (heap, term) pairs.
4. Equivalence of ANS and NNS.

3.2 A Distributed Model (Publication P4)

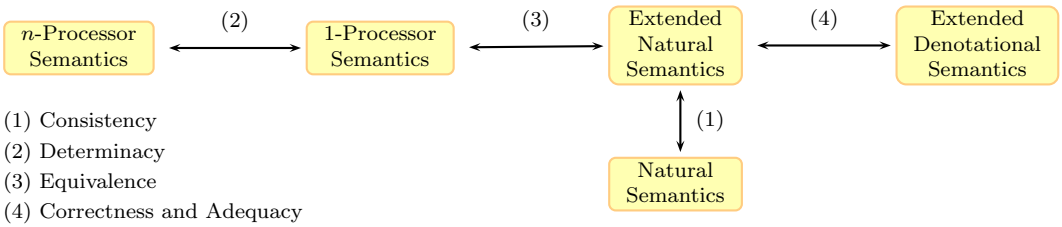
As we have explained in Chapter 1, one main goal of this thesis was to establish some properties (such as correctness and computational adequacy) of an operational semantics for a distributed model with n-processors with respect to a standard denotational semantics. However, while working on the formal proofs for these properties, the problems shown in the previous sections arose.

Based on `Jauja`, the language introduced by Hidalgo-Herrero in [Hid04] and explained in Section 2.5, we have extended the syntax given in [Lau93] with a *parallel application* that produces the creation of new processes. The syntax is given in Figure 3.4. It is a restricted syntax where subexpressions in applications are variables, and the bodies of *let* constructions as well (similarly to the restricted syntax explained in Section 2.4). This restriction simplifies the definition of the semantic rules.

We have revised the operational semantics rules defined by Hidalgo-Herrero in [Hid04] (and briefly described in Section 2.5) and restricted them to *EExp* (n-processors semantics). We have extended, as well, the standard denotational semantics to this calculus by defining the denotation of the parallel application (extended denotational semantics). Finally, we have proved the equivalence between both extensions in terms of correctness and computational adequacy:



This equivalence is not proved directly. We have defined several intermediate semantics and we have given the relation between them, as it is shown in the next schema:



In order to take advantage of Launchbury's work [Lau93], who has established the correctness and computational adequacy for a simpler calculus, we have introduced the *extended natural semantics* (ENS). This semantics is an extension of Launchbury's natural semantics that includes process creations and communications. Processes can only be created if the variables that are needed to evaluate the corresponding application are not blocked. We have two problems: on the one hand, some bindings disappear from the heap due to the variable rule, i.e., information is lost; on the other hand, it is unknown the name to which is bounded the expression that is being evaluated. To solve the first problem, we have extended the heaps. They are now represented as a pair $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$ where the first part stands for the usual heap and the second one describes the blocked bindings, i.e., those that have been demanded either by the variable rule or by the application rules. The solution to the second problem is to keep the name to which an expression is bound. Thus, judgements are now of the form:

$$\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W.$$

Since process creation has to be completed as soon as possible, for λ-abstractions and local declarations rules we require *saturated heaps*, i.e., heaps with no pending process creations.

We have proved the *consistency* of this extension with Launchbury's natural semantics, i.e., in the absence of parallel applications both derivation systems produce the same results:

Theorem 5 (Consistency.) *Let $e \in Exp$. $\Gamma : e \Downarrow \Delta : w$ if and only if $\langle \Gamma, \Gamma^B \rangle : x \mapsto e \Downarrow \langle \Delta, \Gamma^B \rangle : x \mapsto w$, where x is a fresh variable in the derivation and Γ^B is disjoint with respect to Γ and Δ.*

The ENS is defined for a unique processor. To relate it with the n-processor semantics, we have introduced an intermediate small step semantics with process creation and communication, but only one processor, the 1-*processor semantics*.

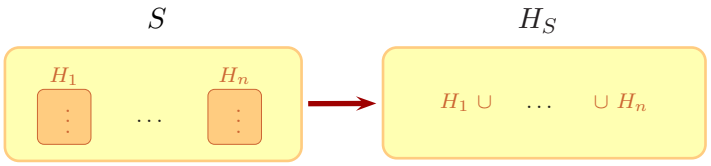


Figure 3.5: Coversion of a distributed system into a heap

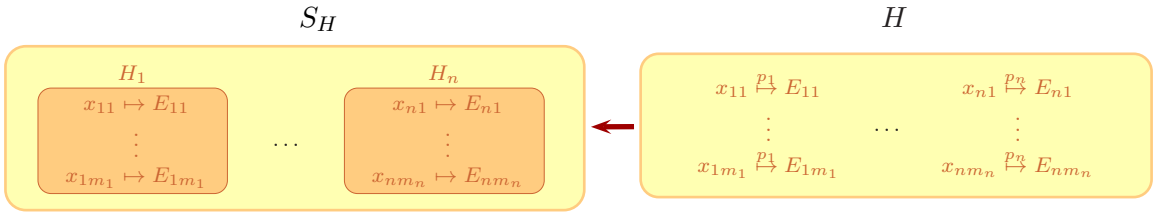


Figure 3.6: Coversion of a non-distributed model into a distributed one

The n-processor semantics represents a distributive model with several processes that evolve, at least, until the value for the main variable is obtained. Recall that there are two types of rules: the local rules that indicate the evolution inside each process; and the global rules to regulate process creations and communications. Each process is represented by a labelled heap. Since in the 1-processor semantics only one binding can be active at any time, we do not need to deal with the two levels. We just have local rules that force an eager process creation and impose an order of evaluation that is compatible with the minimum semantics described in Section 2.5.1.

To prove that the n-processors semantics and the 1-processor semantics produce the same value for the main variable, we need to construct a non-distributed model from a distributed one. For this purpose we build a unique heap that stores all the bindings from the different processes. The potential active bindings become inactive except for one. This unique active binding is given by the function $EB(H)$ that calculates the evolutionary bindings of a heap. This function depends on the semantics that is being used, in this case, the minimum semantics. Therefore, only one binding, that is directly demanded by the main variable, can be active. Figure 3.5 shows the conversion of a distributed system into one heap.

To build a distributed system from a heap of bindings is more difficult. We need to know to which process belongs each binding. This information is added to the bindings in the form of an annotation. These annotations do not interfere with the semantic rules. The schema shown in Figure 3.6 illustrates the conversion of a labelled heap into its associated system.

The determinacy theorem establishes the equivalence between these two semantics when the minimum semantics is considered for the distributed model. Notice that systems are represented as $S = \langle p_i, H_i \rangle_{i=0}^{n}$, where a pair of the form $\langle p_i, H_i \rangle$ represents a process (p_i is the name of the process and H_i its heap).

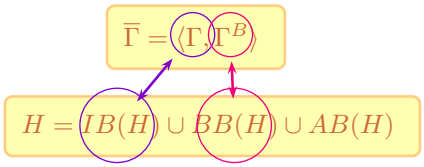


Figure 3.7: Conversion of heaps

Theorem 6 (Determinacy.) *Let $E \in EExp$.*

1. *If $\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* S \Longrightarrow^* S'$, then there exists a computation $\{main \overset{A}{\mapsto} E\} \Longrightarrow_1^* H_S \Longrightarrow_1^* H_{S'}$, where H_S (respectively $H_{S'}$) is the heap constructed from S (respectively S').*

2. *If $\{main \overset{A}{\mapsto} E\} \Longrightarrow_1^* H \Longrightarrow_1^* H'$, then there exists a computation $\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* S_H \Longrightarrow^* S_{H'}$, where S_H (respectively $S_{H'}$) is the process system recovered from heap H (respectively H').*

Now, to prove the equivalence between the 1-processor semantics (small step semantics) and the extended natural semantics (big step semantics), we study the relation between the labelled heaps produced by the 1-processor semantics and those of ENS. The inactive bindings of the heaps obtained by the small step semantics correspond to the first part of the extended heaps, while the blocked bindings correspond to the second part. Moreover, the unique active binding is the expression being evaluated. Conversely, we can construct a labelled heap for the 1-processor semantics from an extended heap. Figure 3.7 shows how to perform this conversion.

Now we are able to establish the equivalence theorem between these two semantics:

Theorem 7 (Equivalence ENS and 1-processor semantics.) *Let $E \in EExp$.*

1. *If $H + \{\theta \overset{A}{\mapsto} E\} \Rightarrow_1^* H' + \{\theta \overset{A}{\mapsto} W\}$, then $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$ where $\overline{\Gamma}$ and $\overline{\Delta}$ are the extended* heaps *associated to H and H' respectively.*

2. *If $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, then $H + \{\theta \overset{A}{\mapsto} E\} \Rightarrow_1^* H' + \{\theta \overset{A}{\mapsto} W\}$ where H and H' are the labelled* heaps *associated to $\overline{\Gamma}$ and $\overline{\Delta}$ respectively.*

After dealing with the equivalence between the different operational semantics, we proceed to prove the *correctness* and *computational adequacy* between the extended versions of the natural semantics and the denotational semantics.

First, we briefly describe how we have extended Abramsky's denotational semantics to denotate channels and parallel applications. The new semantic function is:

$$[\![-]\!]_\rho : \ EExp \cup \mathcal{C}han \to Env \to Value$$

where $\rho \in Env = Var \cup \mathcal{C}han \mapsto Value$ is an environment from identifiers to values. The denotation of a parallel application coincides with the denotation of an application, that is, $[\![x \# y]\!]_\rho = [\![x\ y]\!]_\rho$.

The correctness theorem establishes that the meaning of an expression does not change during its evaluation. Moreover, the size of heaps can only increase (new bindings may

be added). A heap can be also modified by refining its bindings, that is, by updating the closures with the calculated values. We consider *Heap* as the domain of unlabelled heaps, i.e. sets of unlabelled bindings, and $\rho \leq \rho'$ represents the order on environments defined by Launchbury and explained in Section 2.4. We also consider the function $\{\!\{\ldots\}\!\} : \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$ to extend an environment with the bindings of a heap (explained in Section 2.4).

Theorem 8 (Correctness of the extended natural semantics.)
Let $E \in \mathit{EExp} \cup \mathcal{C}\mathit{han}$, $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$, $\overline{\Delta} = \langle \Delta, \Delta^B \rangle \in \mathit{EHeap}$, $y\ \theta \notin \mathit{dom}(\overline{\Gamma})$.
If $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, *then for each environment* ρ, $[\![E]\!]_{\{\!\{\Gamma\}\!\}_\rho} = [\![W]\!]_{\{\!\{\Delta\}\!\}_\rho}$ *and* $\{\!\{\Gamma\}\!\}_\rho \leq \{\!\{\Delta\}\!\}_\rho$.

The computational adequacy determines that an expression reduces to a value if and only if its denotation is not undefined.

Theorem 9 (Adequacy of the extended natural semantics.) *Let* $E \in \mathit{EExp} \cup \mathcal{C}\mathit{han}$, $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle \in \mathit{EHeap}$, $y\ \theta \notin \mathit{dom}(\overline{\Gamma})$. $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, *if and only if* $[\![E]\!]_{\{\!\{\Gamma\}\!\}_\rho} \neq \bot$.

The proofs of these last two theorems are done by following the steps given in [Lau93]. The construction of these proofs have given rise to the study of the results presented in Section 3.1.

Summary of results.

1. Definition of a distributed operational semantics for n processors.
2. Definition of a distributed operational semantics for a unique processor.
3. Extension of the natural semantics with parallel applications.
4. Consistency between the natural semantics and its extension.
5. Equivalence between the 1-processor semantics and the parallel extension.
6. Correctness of the extended natural semantics with respect to an extended denotational semantics.

3.3 Related work

We have already said in Chapter 1 that several works have based on Launchbury's results [Lau93]. Breitner has also noticed, and he has developed in [Bre13, Bre14] a research to prove the correctness and computational adequacy of Launchbury's natural semantics.

Breitner proposes two methods to prove the correctness of the natural semantics in [Bre14]. The first one introduces an equivalent semantic to Launchbury's one where the judgments are of the form $\Gamma : \Gamma' \Downarrow \Delta : \Delta'$, where Γ' and Δ' are ordered sets of bindings. These sets reflect how the evaluation of expressions is being demanded. Therefore, these judgments have more information than those of Launchbury. For instance, when evaluating a variable, the corresponding binding will not disappear from the heap, it will be in Γ'. The idea is similar to the one explained in Section 3.2, where heaps are extended to keep the information of the blocked bindings. As Breitner explains, in the case of Launchbury's semantics, only the topmost binding of the stack can change in each derivation rule. He considers more natural this representation. He also thinks that it can be useful in other cases requiring the modification of the stack, for instance, by an extension with garbage

collection. In the second method, Breitner modifies the denotational semantics and redefines the function that relates the heaps with the environments (explained in Section 2.4), thus the operator for the least upper bound is replaced for a suitable update.

Some of our work related to publication P1 and explained in Section 3.1.1 has been formalized in the proof assistant Isabelle by Breitner [Bre13]. He also proves the computational adequacy of Launchbury's natural semantics without using the alternative version. Recall that Launchbury introduced this version because the heaps of the operational semantics correspond to the environments of the denotational semantics. Breitner solves these problems in the denotational side, and do not need to prove the equivalence between the original Launchbury's natural semantics and the alternative natural semantics.

There is a correspondence between the locally nameless representation and the nominal techniques as Breitner explains in [Bre14], since in both cases the names used in heaps are avoided.

Different methods and relations have been studied over the years to establish the equivalence between semantics. The work of Haeri [Hae09, Hae13] is related to ours in some aspects. He extends Launchbury's language with a `seq` operator and defines a big step semantics for this construction. Based on this semantics, Haeri introduces three notions of equivalence for expressions:

- *Value equivalence:* Two expressions are *value equivalent* if they produce the same value when evaluated in the same initial context;
- *Heap equivalence:* Two expressions are *heap equivalent* if they produce the same final heap when evaluated in the same initial context; and
- *Strict equivalence:* Two expressions are *strictly equivalent* if they produce the same value and final heap when evaluated in the same initial context.

Although a number of interesting properties concerning the semantics are proved using this equivalences, we have not used them because Haeri's semantics differs from Launchbury's natural semantics not only in the extension for `seq`, but also in the rule for local declarations. Following Launchbury, Haeri's rule extends the heap with the local variables in order to evaluate the body of the `let`-expression. But once the value is obtained, the local names are removed from the heap. Therefore, the domains of the initial and final heaps are equal, and their differences come from the updating of closures. We are interested in relating the semantics that Launchbury originally defined (without modifications); we have only changed its representation. Moreover, Haeri's equivalences relate expressions using the same evaluation rules, while we relate expressions (and heaps) evaluated in different reduction systems.

This idea of removing the local names once the body of a `let`-expression is evaluated, is also used by Haeri in [Hae13] for a distributed model that includes the operator #, that represents strict application. Under this operator, the body of the application and its argument are evaluated simultaneously with the same initial context, before the application is performed. A crucial difference with our distributed model, is that Haeri has a very limited form of parallelism, and there is neither notion of process nor of communication.

To deal with the problems derived from α-conversion there exist other alternatives different from the de Bruijn notation an the locally nameless representation explained in Section 2.6. Nowadays the nominal logic [Pit03, GP02] is one of the more popular ones. This logic is based on the fact that the syntactic properties are equivariant, i.e., their validity is invariant under the (*swapping*) of names [Pit03].

This logic does not only present advantages related with α-conversion problems, but it is also implemented in Isabelle, a very well known proof assistant. The nominal package is being used in several studies although it is still a project under development [isa15]. We chose the locally nameless representation because some years ago mutually recursive local declarations could present some problems in nominal Isabelle, and also because Charguéraud's work [Cha11] was very close to our purpose.

The study of Cimini et al. [CMRG12] on structural operational semantics uses some techniques of the nominal logic and works with the notion of bisimilarity. Based on the nominal techniques introduced by Pitts, Gabbay and Urban [GP99, UPG04], the authors develop the Nominal SOS (*Nominal Structural Operational Semantics*), and apply it to the notion of nominal bisimilarity. Later on, they formulate the lazy λ-calculus using the Nominal SOS and prove that it coincides with the original one. They also prove that the nominal bisimilarity coincides with Abramsky's applicative bisimilarity explained in Section 2.3.3.

3.4 Conclusions

In this chapter we have shown the main results that we have obtained in the context of this PhD thesis. We have started by relating two denotational semantics, a standard one and another with resources. This relation has been established by a similarity relation between the values of the domains of the respective semantics. These domains are $D = [D \to D]_\perp$ (as defined by Abramsky) for the standard denotational semantics, and $E = [[C \to E] \to [C \to E]]_\perp$ for the semantics with resources, for which we have construct an initial solution.

In order to prove the equivalence of Launchbury's natural semantics with its alternative version we have introduce two intermediate semantics: one with indirections, and another without update. We have proved the equivalence between the alternative natural semantics and the non-update version. To obtain this result we have defined a preorder, $\succsim_I$, between heaps. To establish this indirection relation we have previously defined some equivalences between terms and heaps. Moreover, we have accomplished this work using a locally nameless representation.

Finally, we have considered a distributed model of the language with parallel applications. To prove the correctness with respect to an extended denotational semantics (that gives meaning to the new expressions), we have introduced two intermediate semantics: a 1-processor semantics and an extension of Launchbury's natural semantics. We have proved the consistency between the natural semantics and its extension, the equivalence between the 1-processor semantics and the extension of the natural semantics, and the correctness of the extended semantics with respect to an extension of the denotational semantics.

We itemize all the results that have been exposed in this chapter.

- Construction of the initial solution of the domain equation $E = [[C \to E] \to [C \to E]]_\perp$, where C stands for the domain of resources.
- Definition of the relation of *similarity* between the values of the domain E and the values of the standard function space $D = [D \to D]_\perp$.
- Application of the previous result to prove the equivalence between the denotational standard semantics and the resourced denotational semantics for the lazy λ-calculus.

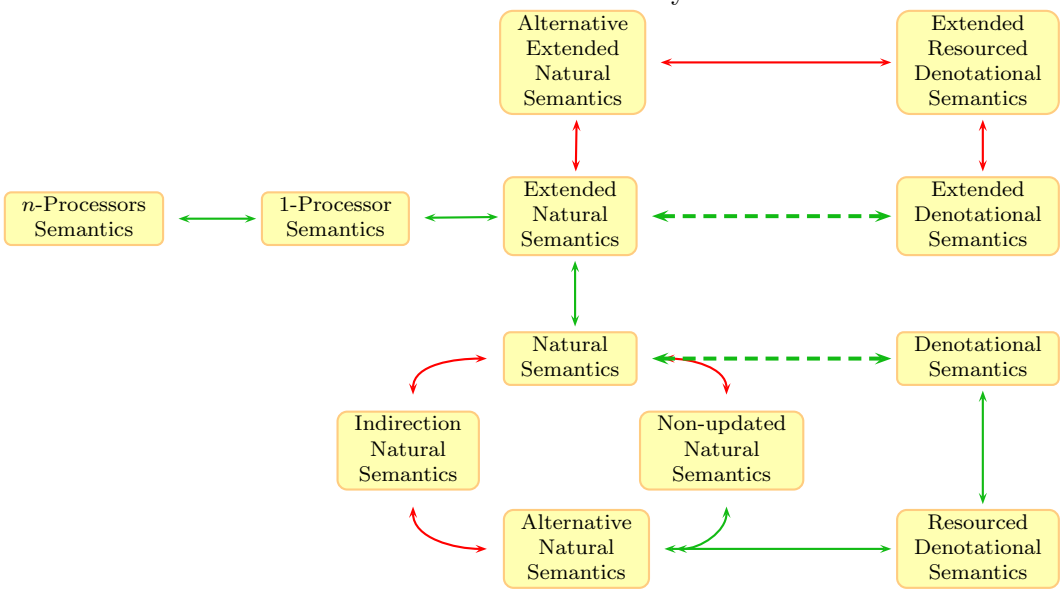
- Locally nameless representation of the λ-calculus extended with recursive local declarations.
- Locally nameless representation of Launchbury's natural semantic rules and its alternative version.
- Properties of the reduction systems (regularity, introduction and renaming).
- Equivalence relation between heaps that define the same free variables, but whose closures may differ in the undefined free variables.
- Preorder that relates two heaps when the former is transformed into the second by the elimination of indirections ($\succsim_I$).
- Extension of the preorder to (heap, term) pairs.
- Equivalence of ANS and NNS.
- Definition of a distributed operational semantics for n processors.
- Definition of a distributed operational semantics for a unique processor.
- Extension of the natural semantics with parallel applications.
- Consistency between the natural semantics and its extension.
- Equivalence between the 1-processor semantics and the parallel extension.
- Correctness of the extended natural semantics with respect to an extended denotational semantics.

Chapter 4

What is left to be done?

We have already explained in Chapter 1 that the aim of this thesis was to study the equivalence between different semantics of a distributed model. There are several avenues for future research.

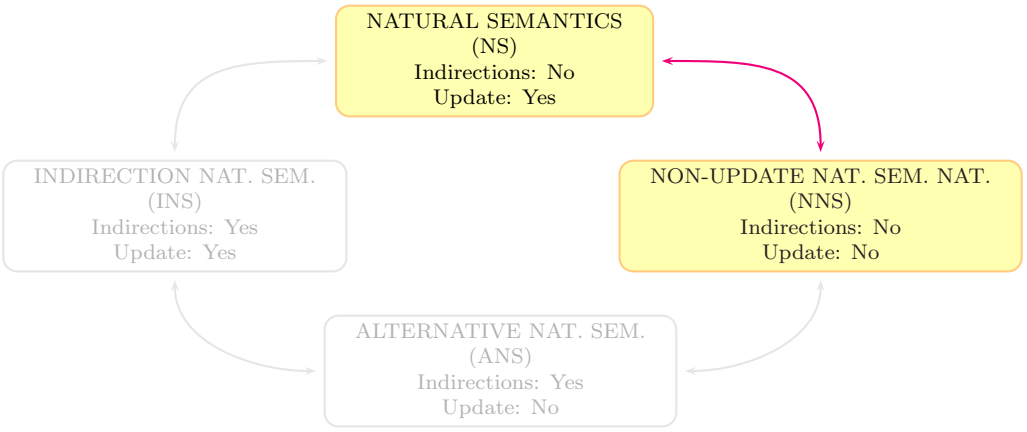
This chapter focuses on the work that we want to achieve in the future. The following schema summarizes in red what is left to finish the study of the distributed model:



We also consider very interesting the implementation of the whole study in a proof assistant. The following sections detail these future research lines: Section 4.1 contains a work that concludes with the equivalence proof of the NS and ANS, this work has been already started; In Section 4.2 we give an alternative proof of the equivalence of NS and ANS; We present the extension of the results to a distributed model in Section 4.3; Finally, Section 4.4 is devoted to the implementation of the results in the proof assistant Coq

4.1 Equivalence NS and NNS (Publication WP1)

First of all, we want to complete the adequacy proof started by Launchbury between his natural semantics and a standard denotational one. To finish with the equivalence of the natural semantics and its alternative version, we have to complete the relation between the natural semantics and the version without update, as the following diagram shows:



We have already started with this study but we do not include it in this PhD because is not finished yet. The work in progress WP1 in Appendix B details what we have already achieved. In this section we just describe the main ideas.

We start with an example to show the differences between the final heaps produced by evaluating an expression with NS and with NNS.

Example 9 *Consider the following term:*

$$t \equiv \begin{array}{l} \texttt{let abs (bvar 0 0), } s\texttt{, app (bvar 0 0) (bvar 0 1)} \\ \texttt{in app (app (app (bvar 0 1) (bvar 0 0)) (bvar 0 0)) (bvar 0 2)} \end{array}$$

where

$$s \equiv \begin{array}{l} \texttt{let abs (bvar 0 0), abs (bvar 0 0), app (bvar 0 0) (bvar 0 1)} \\ \texttt{in app (bvar 0 0) (bvar 0 1)} \end{array}$$

We evaluate the expression with NS ($\Downarrow$) and NNS ($\Downarrow^N$) in the context of an empty heap. We obtain the following results:

$$\{\,\} : t \Downarrow$$

$x_0 \mapsto \texttt{abs (bvar 0 0)}$,
$x_1 \mapsto \texttt{abs (bvar 0 0)}$,
$x_2 \mapsto \texttt{abs (bvar 0 0)}$,
$y_0 \mapsto \texttt{abs (bvar 0 0)}$,
$y_1 \mapsto \texttt{abs (bvar 0 0)}$,
$y_2 \mapsto \texttt{app (fvar } y_0\texttt{) (fvar } y_1\texttt{)}$

$$\{\,\} : t \Downarrow^N$$

$x_0 \mapsto \texttt{abs (bvar 0 0)}$,
$x_1 \mapsto s$,
$x_2 \mapsto \texttt{app (fvar } x_0\texttt{) (fvar } x_1\texttt{)}$,
$y_0 \mapsto \texttt{abs (bvar 0 0)}$,
$y_1 \mapsto \texttt{abs (bvar 0 0)}$,
$y_2 \mapsto \texttt{app (fvar } y_0\texttt{) (fvar } y_1\texttt{)}$,
$z_0 \mapsto \texttt{abs (bvar 0 0)}$,
$z_1 \mapsto \texttt{abs (bvar 0 0)}$,
$z_2 \mapsto \texttt{app (fvar } z_0\texttt{) (fvar } z_1\texttt{)}$

$$w \equiv \texttt{abs (bvar 0 0)}$$

□

Due to the absence of update in NNS, some parts of the computation are repeated. This happens when a variable is demanded more than once during the derivation. Therefore, some bindings are duplicated in the final heap but with different names. There are more differences between the final heaps. For instance, we also observe that some variables are

bound to values in the final heap obtained with NS, while they are not evaluated in the heap obtained with NNS. Therefore, we relate the heaps in two steps: first, we eliminate the duplicated bindings; next, we check that the variables that are bound to different expressions do evaluate to the same value.

We define a *context equivalence* between terms, $\approx^{(\overline{x},\overline{y})}$, to detect the groups of bindings that are similar. Two terms t and t' are equivalent in the contexts $\overline{x}$ and $\overline{y}$ if the terms are equal when closing them in their respective contexts. We use this definition to remove groups of bindings with a similar behavior. The pairs $(\Gamma : t)$ and $(\Gamma' : t')$ are related if after removing from the first one some duplicated group of bindings the resulting (*heap*, term) is still related with $(\Gamma' : t')$. This is formalized in the following definition:

Definition 1 *A heap/term pair* $(\Gamma : t)$ *is* heap-term group related *to a heap/term pair* $(\Gamma' : t')$*, denoted by* $(\Gamma : t) \succsim_G (\Gamma' : t')$*, if:*

$$\text{GR\_EQ\_HT} \quad \frac{}{(\Gamma : t) \succsim_G (\Gamma : t)}$$

$$\text{GR\_GR\_HT} \quad \frac{\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s} \qquad \overline{x} \cap \overline{y} = \emptyset \qquad ((\Gamma, \overline{x} \mapsto \overline{t}) : t)[\overline{x}/\overline{y}] \succsim_G (\Gamma' : t')}{((\Gamma, \overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{s}) : t) \succsim_G (\Gamma' : t')}$$

In the previous definition, $\overline{x} \mapsto \overline{t}$ and $\overline{y} \mapsto \overline{s}$ represent groups of bindings that are equivalent, i.e., $\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s}$. We apply this definition to the heap obtained with NNS ($\Downarrow^N$) in Example 9:

Example 10 *We remove the group* $\overline{z} = [z_0, z_1, z_2]$ *from the final* heap *obtained with NNS in Example 9.*

$x_0 \mapsto$ `abs (bvar 0 0)`,
$x_1 \mapsto s$,
$x_2 \mapsto$ `app (fvar` x_0`) (fvar` x_1`)`,
$y_0 \mapsto$ `abs (bvar 0 0)`,
$y_1 \mapsto$ `abs (bvar 0 0)`,
$y_2 \mapsto$ `app (fvar` y_0`) (fvar` y_1`)`,
$z_0 \mapsto$ `abs (bvar 0 0)`,
$z_1 \mapsto$ `abs (bvar 0 0)`,
$z_2 \mapsto$ `app (fvar` z_0`) (fvar` z_1`)`

$\succsim_G$

$x_0 \mapsto$ `abs (bvar 0 0)`,
$x_1 \mapsto s$,
$x_2 \mapsto$ `app (fvar` x_0`) (fvar` x_1`)`,
$y_0 \mapsto$ `abs (bvar 0 0)`,
$y_1 \mapsto$ `abs (bvar 0 0)`,
$y_2 \mapsto$ `app (fvar` y_0`) (fvar` y_1`)`

Notice that the heap obtained after removing the group has the same size as the heap obtained with NS in Example 9. □

We realize that after removing all the equivalent groups, we have two pairs (heap, term) of the same size. Next we define the update relation, $\sim_U$, where two heaps of the same size are related when the terms bound to the same names are either equal or one of them is a value and the other not. In the latter case we evaluate the term in a given context, and the final (heap, term) pair has to be group related with the (heap, term) pair that contained the term already evaluated.

Definition 2 *Let* $\Gamma, \Gamma', \Delta \in$ *LNHeap. We say that* Γ *is* update related *with* Γ' in the context of Δ*, denoted by* $\Gamma \sim_U^\Delta \Gamma'$*, if*

$$\frac{}{\Gamma \sim_U^\Delta \Gamma} \qquad \frac{\Gamma \sim_U^\Delta \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma, x \mapsto t) \sim_U^\Delta (\Gamma', x \mapsto w')}$$

We are interested in the case where the context of evaluation is precisely the non-updated heap. When the context is not explicitly given we refer to the first heap, i.e., $\Gamma \sim_U \Gamma'$ iff $\Gamma \sim_U^\Gamma \Gamma'$, and we say that Γ is *update related* with Γ'.

We extend this definition to (heap, term) pairs as follows:

$$\frac{\Gamma \sim_U^\Delta \Gamma'}{(\Gamma : t) \sim_U^\Delta (\Gamma' : t)} \qquad \frac{\Gamma \sim_U^\Delta \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma : t) \sim_U^\Delta (\Gamma' : w')}$$

We define the *group-update relation* between (heap, term) pairs, $\succsim_{GU}$, by combining the group relation and the update relation:

$$\frac{(\Gamma : t) \succsim_G (\Delta : s) \quad (\Delta : s) \sim_U (\Gamma' : t') \quad \mathtt{ok}\ \Gamma \quad \mathtt{ok}\ \Gamma' \quad \mathtt{lc}\ t \quad \mathtt{lc}\ t'}{(\Gamma : t) \succsim_{GU} (\Gamma' : t')}$$

We enunciate a theorem indicating that after evaluating the same (heap, term) pair with NS and NNS the final (heap, term) pairs are group-update related. But this theorem has not been proved yet. We suspect that to apply rule induction we will need a generalization of the theorem and some auxiliary lemmas as well, as it has happened with the equivalence between NNS and ANS (Section 3.1.2).

4.2 Equivalence of NS and INS, and of INS and ANS

We consider interesting to close the diagram of the four operational semantics; although this is not necessary, since the adequacy proof will be completed with the result of the previous section. For this, we have to prove the equivalence of NS and ANS but going through INS instead of NNS. We expect that the first part of the proof, the one relating INS and ANS, is similar to the proof that is being developed between NS and NNS. For the second part, i.e., the equivalence of NS and INS, we suspect that it is alike to the study explained in Section 3.1.2, although not exactly the same. Since there is update in both semantics, the indirections, introduced by the application rule, may be updated to a value, so that they are not longer recognizable as indirections (name bound to name). This fact indicates that we have to study not only how to eliminate indirections but also "redundant" bindings.

4.3 Extension to a distributed model

We want to extend the results explained in Chapter 3 and the previous sections of this chapter to the distributed model. We have to follow the same steps in order to achieve this goal. First we have to define an alternative version of the extended natural semantics and to study if we should include indirections when performing a parallel application. We also have to define a resourced denotational semantics for this extension. To conclude the study we have to prove the equivalence between the extended versions of the denotational semantics by applying a "similarity" relation. We have to prove the equivalence between the operational versions of the semantics too. We will probably need to define the corresponding extensions for INS and NNS. A translation of the language and the semantic rules to the locally nameless representation will be done previously.

4.4 Implementation in Coq (Publication WP2)

Automated theorem provers and proof assistants, that have been introduced in Section 2.7, are well known for their reliability in formal proofs. We find of great importance to use one of them to implement the work done in this thesis.

When we started working with Isabelle, the Nominal package was in its origins and the use of mutually recursive local declarations was problematic. Without this nominal package we had to work with the de Bruijn notation. We explained in Section 2.6.1 the disadvantages of this notation.

By contrast, Coq deals well with inductive definitions and allowed to work with recursive local declarations. Moreover, there existed some works using the locally nameless representation [Cha11] with Coq. This was the main reason to choose this proof assistant.

Although the language that we use has applications restricted to variables (Figure 3.2), we decided to work with applications from terms to terms in Coq. To extend the work of Charguéraud [Cha11], we have extended the definition of the language with recursive local declarations represented by a list of terms:

```
Inductive trm :  Type :=
                 | t_bvar : nat− > nat− > trm
                 | t_fvar : var− > trm
                 | t_abs : trm− > trm
                 | t_app : trm− > trm− > trm
                 | t_let : L_trm− > trm− > trm
           with  L_trm :=
                 | nil_Lt : L_trm
                 | cons_Lt : trm− > L_trm− > L_trm.
```

But the induction principle that Coq defines is not transmitted to the list of terms, as it only verifies the property for the main term of the local declarations. This can be seen when checking for the induction principle in Coq:

```
trm_ind :  forall P : trm − > Prop,
           (forall n n0 : nat, P (trm_bvar n n0)) − >
           (forall v : var, P (trm_fvar v)) − >
           (forall t : trm, P t − > P (trm_abs t)) − >
           (forall t : trm, P t − > forall t0 : trm, P t0 − > P (trm_app t t0)) − >
           (forall (l : L_trm) (t : trm), P t − > P (trm_let l t)) − >
           forall t : trm, P t
```

Therefore, we have redefined this principle. There are two mutually recursive properties: `P` for the terms and `P0` for the lists.

```
trm_ind2  forall (P : trm − > Prop) (P0 : L_trm − > Prop),
          (forall n n0 : nat, P (trm_bvar n n0)) − >
          (forall v : var, P (trm_fvar v)) − >
          (forall t : trm, P t − > P (trm_abst)) − >
          (forall t : trm, P t − > forall t0 : trm, P t0 − > P (trm_app t t0)) − >
          (forall l : L_trm, P0 l − > forall t : trm, P t − > P (trm_let l t)) − >
          P0 nil_Ltrm − >
          (forall t : trm, P t − > forall l : L_trm, P0 l − > P0 (cons_Ltrm t l)) − >
          forall t : trm, P t
```

Once we fixed the problems with the induction principle, we have extended some definitions such as variable opening, variable closing, substitution and free variables of a

term. Since the definition of terms is a mutually recursive definition (to build a term we need lists of terms, and to build a list of terms we need terms), all these extensions have to be done for terms and lists simultaneously. For instance, the closure at level k is:

```
Fixpoint close_rec (k : nat) (vs : list var) (t : trm) {struct t} : trm :=
    match t with
    | t_bvar i j => t_bvar i j
    | t_fvar x => if (search_var x vs)
                  then (t_bvar k (pos_elem x vs 0))
                  else (t_fvar x)
    | t_abs t1 => t_abs (close_rec (S k) vs t1)
    | t_app t1 t2 => t_app (close_rec k vs t1) (close_rec k vs t2)
    | t_let ts t => t_let (close_L_rec (S k) vs ts)(close_rec (S k) vs t)
    end
with close_L_rec (k : nat) (vs : list var) (ts : L_trm) {struct ts} : L_trm :=
    match ts with
    | nil_Lt => nil_Lt
    | cons_Lt t ts => cons_Lt (close_rec k vs t) (close_L_rec k vs ts)
    end.
```

Similarly, the predicate of local closure has to be done on terms and lists of terms simultaneously:

```
Inductive lc : trm -> Prop :=
    | lc_var : forall x, lc (t_fvar x)
    | lc_app : forall t1 t2, lc t1 -> lc t2 -> lc (t_app t1 t2)
    | lc_abs : forall L t, (forall x, x ∉ L -> lc (open t (cons x nil))) -> lc (t_abst)
    | lc_let : forall L t ts, (forall xs, xs ∉ L -> (lc_list (opens (cons_Lt t ts) xs)))
                              -> lc (t_let ts t)
with lc_list : L_trm -> Prop :=
    | lc_list_nil : lc_list (nil_Lt)
    | lc_list_cons : forall t ts, lc t -> lc_list ts -> lc_list(cons_Lt t ts).
```

We also have defined the bindings, the heaps and several functions on heaps such as the domain, the names, the substitution and the predicate of well-defined heaps explained in Section 3.1.2. But these definitions have to be revised since they seem to be inadequate to work with the semantic rules and further proofs. For instance, we have defined a heap as a set of bindings, but we were told that it would be better to define it as a partial function.

This work was postponed but not forgot. Working with proof assistants is not easy. Some proofs, that seem to be trivial when developing them by hand, turn to be complicate when performing them in a machine. This part of the research requires a lot of time, and we expect to finish it in the future.

There is still a lot of work to develop, but we think that what we exposed in Chapter 3 is enough to conform a thesis.

Bibliografía

[Abr90] S. Abramsky. *Research Topics in Functional Programming*, chapter 4: The lazy lambda calculus, pages 65–117. Ed. D. A. Turner. Addison Wesley, 1990.

[Abr91] S. Abramsky. Domain theory in logical form*. *Annals of Pure and Applied Logic*, 51(1-2):1 – 77, 1991.

[ACP+08] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages, POPL'08*, pages 3–15. ACM Press, 2008.

[agd14] Agda. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2014. Accedido 30-10-2014.

[AJ94] S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science (Vol. 3)*, pages 1–168. Oxford University Press, 1994.

[AO93] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BB00] H. Barendregt and E. Barendsen. Introduction to lambda calculus, 2000.

[BKT00] C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming ICFP'00*, pages 162–173. ACM Press, 2000.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report 96/10, Reihe Informatik, FB Mathematik, Philipps-Universität Marburg, Germany, URL `http://www.mathematik.uni-marburg.de/~eden/`, 1996.

[Bre13] J. Breitner. The correctness of Launchbury's natural semantics for lazy evaluation. *Archive of Formal Proofs*, January 2013. `http://afp.sf.net/entries/Launchbury.shtml`, Formal proof development.

[Bre14] J. Breitner. The correctness of Launchbury's natural semantics for lazy evaluation. *arXiv:1405.3099v1*, 2014.

[Cha11] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011.

[CMRG12] M. Cimini, M. R. Mousavi, M. A. Reniers, and M. J. Gabbay. Nominal SOS. *Electronic Notes in Theoretical Computer Science*, 286:103–116, 2012.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[coq14] The Coq proof assistant. `http://coq.inria.fr/`, 2014. Accedido 30-10-2014.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[ede14] Eden. `http://www.mathematik.uni-marburg.de/~eden/`, 2014. Accedido 15-12-2014.

[erl14a] Erlang solutions. `https://www.erlang-solutions.com/industries`, 2014. Accedido 30-10-2014.

[erl14b] What is Erlang. `https://www.erlang.org/faq/introduction.html`, 2014. Accedido 30-10-2014.

[Esp14] Real Academia Española. Diccionario de la lengua española (22.ª edición). `http://www.rae.es/recursos/diccionarios/drae`, 2014. Accedido 30-10-2014.

[Geu09] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, February 2009.

[Gor94] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, HUG '93, pages 413–425. LNCS 780, Springer-Verlag, 1994.

[GP99] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.

[GP02] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2002.

[Hae09] S. H. Haeri. Reasoning about selective strictness. Master's thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, 2009.

[Hae13] S. H. Haeri. A new operational semantics for distributed lazy evaluation. Technical Report TR2013-1, Institute for Software Systems, Technische Universität Hamburg, 2013.

[has14a] Haskell in industry. `https://www.haskell.org/haskellwiki/Haskell_in_industry`, 2014. Accedido 30-10-2014.

[has14b] The haskell programming language. `https://www.haskell.org/haskellwiki/Haskell`, 2014. Accedido 30-10-2014.

[has15] Lazy evaluation. `https://wiki.haskell.org/Lazy_evaluation`, 2015. Accedido 14-05-2015.

[Hid04] M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo*. PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[HO02] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. In *International Workshop on Constructive Methods for Parallel Programming, CMPP'02*, pages 63–79. Technische Universitaet Berlin, Germany, 2002.

[isa14] Isabelle. `http://isabelle.in.tum.de/`, 2014. Accedido 30-10-2014.

[isa15] Nominal Isabelle. `http://isabelle.in.tum.de/nominal/`, 2015. Accedido 22-01-2015.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, (POPL'93)*, pages 144–154. ACM Press, 1993.

[Ler07] X. Leroy. A locally nameless solution to the POPLmark challenge. Technical report, INRIA, January 2007.

[Loo99] R. Loogen. *Research Directions in Parallel Functional Programming*, chapter 3: Programming Language Constructs, pages 63–92. Eds. K. Hammond and G. Michaelson. Springer-Verlag, 1999.

[LOP05] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[mir15] Miranda. `http://http://miranda.org.uk/`, 2015. Accedido 14-05-2015.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

[NH09] K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *CoRR*, abs/0907.4640, 2009.

[OSV10] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2010.

[Pey03] S. L. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[Pit13] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

[pvs14] PVS specification and verification system. `http://pvs.csl.sri.com/`, 2014. Accedido 30-10-2014.

[Rea89] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[Sco73] D. Scott. Models for various type-free calculi. *Logic, Methodology, and Philosophy of Science IV*, pages 157–187, 1973.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[SGHHOM10] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. *Trends in Functional Programming*, volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pages 65–80. Intellect, 2010.

[SGHHOM11] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. Relating function spaces to resourced function spaces. In *ACM Symposium on Applied Computing*, SAC '11, pages 1301–1308. ACM Press, 2011.

[SGHHOM12a] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A formalization in Coq of Launchbury's natural semantics for lazy evaluation. In *Jornadas sobre Programación y Lenguajes, PROLE'12*, pages 15–29, 2012.

[SGHHOM12b] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. In *International Colloquium Theoretical Aspects of Computing, ICTAC'12*, pages 105–119. LNCS 7521, Springer-Verlag, 2012.

[SGHHOM12c] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. Technical Report 01/12, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid, 2012. http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-1-12.pdf.

[SGHHOM13] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. The role of indirections in lazy natural semantics (extended version). Technical Report 13/13, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid, 2013. http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/TR-13-13.pdf.

[SGHHOM14a] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. Launchbury's semantics revisited: On the equivalence of context-heap semantics. In *Jornadas sobre Programación y Lenguajes, PROLE'14*, pages 203–217, 2014.

[SGHHOM14b] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. The role of indirections in lazy natural semantics. In *PSI*, 2014. Pendiente de publicación en LNCS, Springer-Verlag.

[THLP98] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Funcional Programming*, 8(1):23–60, 1998.

[TLP03] P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4+5):469–510, 2003.

[UBN07] C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *International Conference on Automated Deduction*, pages 35–50. LNCS 4603, Springer-Verlag, 2007.

[UPG04] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

[vEdM07] M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud University Nijmegen, 2007.

[wha14] Código abierto de whatsapp. `https://www.whatsapp.com/opensource/`, 2014. Accedido 30-10-2014.

[Wie06] F. Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*. LNCS 3600, Springer-Verlag, 2006.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

Parte III

Publicaciones

Capítulo 5

Publicaciones

Este capítulo contiene las cuatro publicaciones principales que forman la tesis. Al igual que se hizo en el Capítulo 3, las publicaciones aparecen de forma temática en lugar de cronológicamente.

En primer lugar presentamos el artículo P1 [SGHHOM11] sobre la construcción de dominios, semánticas denotacionales y su relación mediante bisimilitud. El trabajo fue presentado en el 26^{th} *Symposium on Applied Computing* (SAC) que tuvo lugar en Taichung (Taiwán) en septiembre de 2011, y fue publicado por ACM en las actas del simposium.

A continuación hay un bloque de dos artículos relacionados con la demostración de la equivalencia entre las dos semánticas operacionales presentadas por Launchbury [Lau93]. La primera publicación de este bloque, P2 [SGHHOM12b], contiene la representación localmente sin nombres de la semántica natural y la demostración de diversas propiedades relacionadas con esta representación. Esta representación facilita las demostraciones de los resultados que se exponen en la segunda publicación. El artículo sobre la representación localmente sin nombres se presentó en el 9^{th} *International Colloquium on Theoretical Aspects of Computing* (ICTAC) en Bangalore (India) en septiembre de 2012 y se publicó en el volumen 7521 de la serie LNCS de Springer-Verlag. En cuanto al segundo artículo de este bloque, P3 [SGHHOM14b], se presentó en la novena edición de *Perspectives of System Informatics* (PSI) en San Petersburgo (Rusia) en junio de 2014 y se publicó en el volumen 8974 de la serie LNCS de Springer-Verlag. En este artículo se hace un estudio sobre el papel que juegan las indirecciones a lo largo de una evaluación, y se establece una relación entre los contextos y los valores obtenidos al evaluar un mismo término con dos semánticas distintas, una de las cuales introduce indirecciones.

Cierra el capítulo la publicación P4 [SGHHOM10] en la que se hace un estudio de distintas semánticas sobre un modelo distribuido. Este trabajo fue presentado en junio de 2009 en el 10^{th} *Trends in Functional Programming* celebrado en Komarno (Eslovaquia), y posteriormente seleccionado para su publicación en el volumen 10 de *Trends in Functional Programming*, Intellect.

This chapter contains the main publications that form this thesis. As we have done in Chapter 3, these publications are presented according to their thematic.

The first paper, P1 [SGHHOM11]*, is about domain construction, denotational semantics and their relation through bisimilarity. This was presented in the* 26^{th} Symposium on Applied Computing *(SAC) in Taichung (Taiwan) in September 2011. This was published by ACM in the proceedings of the symposium.*

Next, there are two papers devoted to the missing proof of Launchbury's work. The

first one, P2 [SGHHOM12b], *focuses on the locally nameless representation of the natural semantics for lazy evaluation. Some properties of this representation are proved as well. This work is useful for the development of the proofs of the second paper. This paper was presented in the* 9^{th} International Colloquium on Theoretical Aspects of Computing *(ICTAC) in Bangalore (India) in September 2012 and was published in the volume 7521 of the LNCS by Springer-Verlag. The second work,* P3 [SGHHOM14b], *was presented in the* 9^{th} Perspectives of System Informatics *(PSI) in Saint Petersburg (Russia) in June 2014 and was published in the volume 8974 of the LNCS by Springer-Verlag. In this work we study the role of indirections during the evaluation. We define a relation between the context and value obtained when an expression is evaluated with two different semantics.*

The last publication, P4 [SGHHOM10], *is dedicated to a distributed model, where the relation between several semantics is shown. This paper was presented in the* 10^{th} Trends in Functional Programming *in Komarno (Slovakia), and it was published in volume 10 of* Trends in Functional Programming *by Intellect.*

Relating function spaces to resourced function spaces

Lidia Sánchez-Gil
Universidad Complutense de Madrid
lidiasg@mat.ucm.es

Mercedes Hidalgo-Herrero
Universidad Complutense de Madrid
mhidalgo@edu.ucm.es

Yolanda Ortega-Mallén
Universidad Complutense de Madrid
yolanda@sip.ucm.es

ABSTRACT

In order to prove the computational adequacy of the (operational) natural semantics for lazy evaluation with respect to a standard denotational semantics, Launchbury defines a resourced denotational semantics. This should be equivalent to the standard one when given infinite resources, but this fact cannot be so directly established, because each semantics produces values in a different domain. The values obtained by the standard semantics belong to the usual lifted function space $D = [D \to D]_\perp$, while those produced by the resourced semantics belong to $[C \to E]$ where E satisfies the equation $E = [[C \to E] \to [C \to E]]_\perp$ and C (the domain of resources) is a countable chain domain defined as the least solution of the domain equation $C = C_\perp$.

We propose a way to relate functional values in the standard lifted function space to functional values in the corresponding resourced function space. We first construct the initial solution for the domain equation $E = [[C \to E] \to [C \to E]]_\perp$ following Abramsky's construction of the initial solution of $D = [D \to D]_\perp$. Then we define a "similarity" relation between values in the constructed domain and values in the standard lifted function space. This relation is inspired by Abramsky's applicative bisimulation.

Finally we prove the desired equivalence between the standard denotational semantics and the resourced semantics for the lazy λ-calculus.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*

Keywords

Domain theory, denotational semantics, λ-calculus.

1. MOTIVATION

There is a mismatch between the pure λ-calculus in the standard theory [4] and the practice of functional programming, or more precisely, the lazy functional languages. This situation has been pointed out clearly by Abramsky in his seminal paper [1], where he proposes a "lazy" theory based on the notion of *applicative transition systems* and introduces a suitable domain equation, i.e., $D = [D \to D]_\perp$, which has a nontrivial initial solution that constitutes a canonical model for the family of lazy languages. The construction of this initial solution is detailed in [2]. We adapt this construction to the case where resources are added to the computational model, so that we obtain the initial solution for the domain equation $E = [[C \to E] \to [C \to E]]_\perp$, where C is a countable chain domain defined as the least solution of the domain equation $C = C_\perp$.

"Resourced" domains like E enable to define semantics that restrict the number of available resources. For instance, let us consider that only 60 seconds of CPU time are available and that each reduction needs one second; thus after 60 reductions the system gets blocked and the semantic value is $\perp$. Another example: Boudol, Curien and Lavatelli present in [5] a λ-calculus with resources to provide a control on the substitution process, so that a computation gets in deadlock when there are not enough resources to carry out all the substitutions. For this they use a different domain equation, namely $D = [\mathcal{M}(D) \to D]_\perp$, where $\mathcal{M}(D)$ stands for multisets of D.

The resourced domain E was used by Launchbury in [7] to define a (denotational) resourced semantics that was introduced to prove the computational adequacy of his (operational) natural semantics for lazy evaluation. In that resourced semantics each syntactic level (in the term to be evaluated) requires the consumption of one resource. This mimics the derivation process for the natural semantics, where one operational rule is applied to eliminate each syntactic level. Hence, denotations that consume a finite number of resources correspond to finite derivations. Nevertheless, the construction of the initial solution for the domain equation $E = [[C \to E] \to [C \to E]]_\perp$ is not detailed in Launchbury's work, and we have found it described nowhere.

Our motivation for constructing an initial solution for $E = [[C \to E] \to [C \to E]]_\perp$ (let us call it *CValue*) is to define a correspondence between its elements and those of the lifted function space $D = [D \to D]_\perp$ (let us call it *Value*). This correspondence is necessary to prove that, provided an unbounded number of resources, the resourced semantics equals the standard denotational semantics. In [7] Launchbury simply states that both semantics compute the same values, but this is not exact because the values produced by the standard semantics belong to *Value*, while

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

those obtained by the resourced semantics belong to *CValue*. How can be related functional values belonging to different semantic domains? The obvious answer is to observe their behaviour when applying them to any argument; but arguments and results are functional values too, thus some kind of recursive definition is needed. We are inspired by the *applicative bisimulation* defined by Abramsky in [1] as the limit of a sequence of relations, each one allowing to observe the applicative behaviour of functions until some fixed depth. But in Abramsky's applicative bisimulation the "experiments" are common to both sides of the simulation — namely, they are syntactic terms— while in our case, they are again values belonging to different domains. Therefore, our version of the applicative bisimulation is in some way more general, and it can be useful for the study of simulations between distinct sets.

Summarizing, the contributions of this paper are: (1) the construction of the initial solution for the domain equation $E = [[C \to E] \to [C \to E]]_\perp$, where C represents resources and satisfies the equation $C = C_\perp$, (2) the definition of a "similarity" relation between values in the constructed domain and values in the standard lifted function space $D = [D \to D]_\perp$, and (3) the proof of the equivalence between a standard denotational semantics and a resourced semantics for the lazy λ-calculus.

These results allow us to obtain a proof for the computational adequacy of Launchbury's natural semantics. As we have pointed out above, Launchbury just assumes that values obtained from both denotational semantics can be directly related, but the adequacy proof is not detailed. To the best of our knowledge, no complete and detailed proof of this adequacy has been published before. When trying to reproduce it we have discovered that it is not a straightforward proof, and that several subtle difficulties arise.

The adequacy result is of interest since Launchbury's natural semantics has been often cited and has inspired many other works as well as several extensions of his semantics, where the corresponding adequacy proofs have been obtained by adapting Launchbury's proof scheme, e.g., [14, 3, 15, 9, 11]. Moreover, the resourced function space and the similarity relation defined here can be useful in other contexts.

The paper is organized as follows: In Section 2 we define the resourced function space and we detail the construction of the initial solution of the corresponding domain equation. In Section 3 we define a similarity relation between the values of the standard function space and those of the resourced function space. In Section 4 we describe a standard and a resourced denotational semantics for a lazy λ-calculus, and we prove their equivalence. The last section is devoted to the conclusions and the outline of future work.

2. A RESOURCED FUNCTION SPACE

A *partially ordered set* $(D, \sqsubseteq_D)$ is a *directed-complete partial order* (dcpo) if every directed subset has a supremum. A *complete partial order* (cpo) is a dcpo with a least element. For simplicity a cpo is usually represented by just its set.

A function $f : D \to D'$ between cpos D and D' is continuous if it maps directed sets to directed sets while preserving their suprema.

Following Scott's domain theory [6], the *function space* $[D \to D']$, being $(D, \sqsubseteq_D)$ and $(D', \sqsubseteq_{D'})$ cpos, is defined as $[D \to D'] \stackrel{\text{def}}{=} \{f \mid f : D \to D' \text{ is continuous}\}$, where the order is $f \sqsubseteq_{[D\to D']} g \stackrel{\text{def}}{=} \forall d \in D. f(d) \sqsubseteq_{D'} g(d)$.

The *lifted* construction of a cpo $(D, \sqsubseteq_D)$ is defined as:

$$\begin{array}{lll} D_\perp & \stackrel{\text{def}}{=} & \{\perp\} \cup \{\lfloor d \rfloor \mid d \in D\}, \\ x \sqsubseteq_{D_\perp} y & \stackrel{\text{def}}{=} & x = \perp \vee (x = \lfloor d \rfloor \wedge y = \lfloor d' \rfloor \wedge d \sqsubseteq_D d'). \end{array}$$

The function $\lfloor - \rfloor$ verifies that $\lfloor d \rfloor = \lfloor d' \rfloor \Rightarrow d = d'$ and $\perp \neq \lfloor d \rfloor$ for all $d, d' \in D$.

Two functions, $\mathsf{up}_D : D \to D_\perp$ and $\mathsf{dn}_D : D_\perp \to D$, relate the cpo D to its lifted version $D_\perp$ and viceversa:

$$\begin{array}{lll} \mathsf{up}_D(d) & \stackrel{\text{def}}{=} & \lfloor d \rfloor, \\ \\ \mathsf{dn}_D(\perp) & \stackrel{\text{def}}{=} & \perp_D, \\ \mathsf{dn}_D(\lfloor d \rfloor) & \stackrel{\text{def}}{=} & d. \end{array}$$

They verify that $\mathsf{dn}_D \circ \mathsf{up}_D = id_D$ and $id_{D_\perp} \sqsubseteq \mathsf{up}_D \circ \mathsf{dn}_D$.

There are different ways of realising $\lfloor - \rfloor$, all of them leading to isomorphic constructions. In the following we will consider that $\lfloor d \rfloor = d$.

Let $f : D \to D'$, it can be extended to $f_\perp : D_\perp \to D'_\perp$:

$$\begin{array}{lll} f_\perp(\perp) & \stackrel{\text{def}}{=} & \perp \text{ , and} \\ f_\perp(\mathsf{up}_D(d)) & \stackrel{\text{def}}{=} & \mathsf{up}_{D'}(f(d)) \text{ for } d \in D. \end{array}$$

The construction of the initial solution of the domain equation $D = [D \to D]_\perp$ is detailed in [2]. The objective of the present section is to adapt this construction to obtain the initial solution for the domain equation $E = [[C \to E] \to [C \to E]]_\perp$, being C the least solution of the domain equation $C = C_\perp$. The elements of C are represented as $\perp$, $S(\perp)$, $S^2(\perp) = S(S(\perp))$, ..., $S^n(\perp)$, ..., being $S^\infty(\perp)$ its limit element, where S stands for a successor-like function. In the following, to simplify, we abbreviate $S^\infty(\perp)$ to S^∞.

The initial solution for the domain equation $E = [[C \to E] \to [C \to E]]_\perp$ is constructed by finite approximations. Each pair injection/projection between successive approximations constitutes an embedding.

2.1 Embeddings

Let D and D' be cpos. An *embedding* of D into D' is a pair of continuous maps $\langle i, j \rangle$ such that $D \stackrel{i}{\rightarrowtail} D' \stackrel{j}{\twoheadrightarrow} D$, and $i \circ j \sqsubseteq id_{D'}$ and $j \circ i = id_D$, where $\rightarrowtail$ and $\twoheadrightarrow$ stand for an injection and a projection respectively.

Given an embedding of D into D', it is possible to build other embeddings between cpos constructed from D and D'.

Consider the composition application over cpos X, Y and Z with the following signature:

$$\circ : [Y \to Z] \times [X \to Y] \to [X \to Z].$$

Then, for any $f \in [Y \to Z]$ and $g \in [X \to Y]$, we can define the function sections:

$$\begin{array}{l} (f \circ) : [X \to Y] \to [X \to Z] \text{ and} \\ (\circ\, g) : [Y \to Z] \to [X \to Z], \end{array}$$

so that $(f \circ)(h) \stackrel{\text{def}}{=} (f \circ h)$ and $(\circ\, g)(h) \stackrel{\text{def}}{=} (h \circ g)$ for a function h of the appropiate type in each case.

LEMMA 1. *Let $\langle i, j \rangle$ be an embedding of D into D'. For any cpo X, $\langle (i \circ), (j \circ) \rangle$ is an embedding of $[X \to D]$ into $[X \to D']$.*

PROOF. We have to show that $(j \circ) \circ (i \circ) = id_{[X\to D]}$ and $(i \circ) \circ (j \circ) \sqsubseteq id_{[X\to D']}$.

Let $f \in [X \to D]$, we have $(j \circ) \circ (i \circ)(f) = (j \circ) \circ (i \circ f) = (j \circ i) \circ f = id_D \circ f = f$, by associativity of $\circ$.

Similarly, if $g \in [X \to D']$ then $(i \circ) \circ (j \circ)(g) = (i \circ) \circ (j \circ g) = (i \circ j) \circ g \sqsubseteq id_{D'} \circ g = g$. $\square$

As a consequence, from an embedding of D into D', an embedding between the resourced domains $[C \to D]$ and $[C \to D']$ can be built by choosing X as the countable chain domain C defined above.

COROLLARY 2. *Let $\langle i, j\rangle$ be an embedding of D into D'. Then $\langle i^C, j^C\rangle$ is an embedding of $[C \to D]$ into $[C \to D']$, which is defined as $i^C(a)(c) \stackrel{def}{=} i(a(c))$ and $j^C(b)(c) \stackrel{def}{=} j(b(c))$, where $a \in [C \to D]$, $b \in [C \to D']$ and $c \in C$.*

PROOF. Notice that $i^C = (i \circ)$ and $j^C = (j \circ)$; then we use Lemma 1. □

Given an embedding between two cpos, an embedding between their lifted domains can be easily obtained.

LEMMA 3. *Let $\langle i, j\rangle$ be an embedding of D into D'. The pair $\langle i_\perp, j_\perp\rangle$ is an embedding of $D_\perp$ into $D'_\perp$.*

PROOF. Trivial from the definition of $i_\perp$ and $j_\perp$. □

Next we define how to build an embedding between function spaces from an embedding between their ground cpos.

LEMMA 4. *Let $\langle i, j\rangle$ be an embedding of D into D'. Then $\langle i^\to, j^\to\rangle$ is an embedding of $[D \to D]$ into $[D' \to D']$, where:*

$$\begin{array}{lll} i^\to & \stackrel{def}{=} & (i \circ) \circ (\circ\, j), \text{ and} \\ j^\to & \stackrel{def}{=} & (j \circ) \circ (\circ\, i). \end{array}$$

PROOF. Since $\langle i, j\rangle$ is an embedding of D into D', it is verified that $j \circ i = id_D$ and $i \circ j \sqsubseteq id_{D'}$. We have to check that $j^\to \circ i^\to = id_{[D\to D]}$ and $i^\to \circ j^\to \sqsubseteq id_{[D'\to D']}$. We use again the associativity of $\circ$.

Let $f \in [D \to D]$ and $g \in [D' \to D']$:

$$\begin{array}{rcl} (i^\to \circ j^\to)(g) & = & ((i \circ) \circ (\circ\, j)) \circ ((j \circ) \circ (\circ\, i))(g) \\ & = & ((i \circ) \circ (\circ\, j))(j \circ g \circ i) \\ & = & i \circ (j \circ g \circ i) \circ j \\ & = & (i \circ j) \circ g \circ (i \circ j) \\ & \sqsubseteq & id'_D \circ g \circ id'_D \\ & = & g. \end{array}$$

Similarly, it is proved that $(j^\to \circ i^\to)(f) = f$. □

Now we combine the two previous constructions to obtain an embedding between lifted function spaces.

COROLLARY 5. *Let $\langle i, j\rangle$ be an embedding of D into D'. The pair $\langle i^*, j^*\rangle$, which is defined as $i^* \stackrel{def}{=} (i^\to)_\perp$ and $j^* \stackrel{def}{=} (j^\to)_\perp$, is an embedding of $[D \to D]_\perp$ into $[D' \to D']_\perp$.*

PROOF. Is a consequence of Lemmas 3 and 4. □

This corollary is equivalent to Lemma 4.1.1. given by Abramsky and Ong in [2]; although their definition of $\langle i^*, j^*\rangle$ is more cumbersome because the lifting and function space constructions are combined in a unique step.

These results also can be easily obtained by considering embeddings as Galois connections [8].

We are now ready to define an embedding of $[[C \to D] \to [C \to D]]_\perp$ into $[[C \to D'] \to [C \to D']]_\perp$. We start with an embedding $\langle i, j\rangle$ of D into D'; by Corollary 2 we obtain $\langle i^C, j^C\rangle$, an embedding of $[C \to D]$ into $[C \to D']$. Finally we get an embedding $\langle i^{C*}, j^{C*}\rangle$ of $[[C \to D] \to [C \to D]]_\perp$ into $[[C \to D'] \to [C \to D']]_\perp$ by applying Corollary 5 (see Figure 1).

The embedding $\langle i^{C*}, j^{C*}\rangle$ is used in the next subsection to construct the initial solution of $E = [[C \to E] \to [C \to E]]_\perp$.

2.2 Construction of the initial solution

We represent the finite aproximations of E by $\{E_n\}_{n\in\mathbb{N}}$, defined inductively as $E_0 \stackrel{\text{def}}{=} \{\perp_{E_0}\}$ and $E_{n+1} \stackrel{\text{def}}{=} [[C \to E_n] \to [C \to E_n]]_\perp$. Intuitively, E_n contains the (resourced) functions that can be applied until level n at most. Hence, E_0 contains only the bottom value, indicating that it cannot be applied. While E_1 has two elements, i. e., $E_1 = \{\perp_{E_1}, e_1\}$ where e_1 is the function that maps a_0 into a_0, being $A_0 = [C \to E_0] = \{a_0\}$ such that $\forall n \in \mathbb{N}^\infty (= \mathbb{N} \cup \{\infty\}).a_0(S^n(\perp)) = \perp_{E_0}$. Therefore, e_1 can be applied in a first level (to $a_0 \in A_0$) but the resulting value (a_0) instantiated with any amount of resources produces $\perp_{E_0}$, which cannot be applied.

Let $\langle i_0, j_0\rangle$ be an embedding of E_0 into E_1, where i_0 and j_0 are defined as follows:

$$\begin{array}{rcccc} i_0 & : & E_0 & \longrightarrow & E_1 \\ & & \perp_{E_0} & \mapsto & \perp_{E_1} \end{array} \qquad \begin{array}{rcccc} j_0 & : & E_1 & \longrightarrow & E_0 \\ & & \perp_{E_1} & \mapsto & \perp_{E_0} \\ & & e_1 & \mapsto & \perp_{E_0} \end{array}$$

The embeddings of E_{n+1} into E_{n+2}, i. e. $E_{n+1} \overset{i_{n+1}}{\rightarrowtail} E_{n+2} \overset{j_{n+1}}{\twoheadrightarrow} E_{n+1}$, are defined as $\langle i_{n+1}, j_{n+1}\rangle \stackrel{\text{def}}{=} \langle i_n^{C*}, j_n^{C*}\rangle$, that is,

$$\begin{array}{rcl} i_{n+1}(e_{n+1}) = i_n^{C*}(e_{n+1}) & = & (i_n^{C\to})_\perp(e_{n+1}) \\ & = & \begin{cases} \perp_{E_{n+2}} & \text{if } e_{n+1} = \perp_{E_{n+1}} \\ i_n^C \circ e_{n+1} \circ j_n^C & \text{if } e_{n+1} \neq \perp_{E_{n+1}} \end{cases} \end{array}$$

$$\begin{array}{rcl} j_{n+1}(e_{n+2}) = j_n^{C*}(e_{n+2}) & = & (j_n^{C\to})_\perp(e_{n+2}) \\ & = & \begin{cases} \perp_{E_{n+1}} & \text{if } e_{n+2} = \perp_{E_{n+2}} \\ j_n^C \circ e_{n+2} \circ i_n^C & \text{if } e_{n+2} \neq \perp_{E_{n+2}} \end{cases} \end{array}$$

$\langle E_n, j_n\rangle_{n\in\mathbb{N}}$ forms an inverse system of cpos and we take its inverse limit [13] as E, i.e.,

$$\begin{array}{l} E = \lim_{\leftarrow}\langle E_n, j_n\rangle \\ \quad = \{\langle e_n : n \in \mathbb{N}\rangle \mid e_n \in E_n \wedge j_n(e_{n+1}) = e_n\} \subseteq \prod_{n\in\mathbb{N}} E_n. \end{array}$$

where the tuple $\langle e_n : n \in \mathbb{N}\rangle = \langle e_0, e_1, \ldots, e_n, \ldots\rangle$ represents an element $e \in E$ by its approximations in each layer.

The embeddings of E_n into E_{n+1} can be generalized to relate E_m to E_n, for any $n, m \in \mathbb{N}^\infty$.

We define the functions $\phi^E_{m,n} : E_m \to E_n$ as follows:

$$\begin{array}{ll} m = n & \phi^E_{n,n} \stackrel{\text{def}}{=} id_{E_n}, \\ m > n & \phi^E_{m,n} \stackrel{\text{def}}{=} \phi^E_{m-1,n} \circ j_{m-1}, \\ m < n & \phi^E_{m,n} \stackrel{\text{def}}{=} i_{n-1} \circ \phi^E_{m,n-1}. \end{array}$$

The *n-projection* $\phi^E_{\infty,n} : E \to E_n$ is defined as the limit of the projections $\phi^E_{m,n}$. For simplicity we write ψ^E_n for $\phi^E_{\infty,n}$. Similarly, ϕ^E_n represents the *n-injection* $\phi^E_{n,\infty} : E_n \to E$. By construction $\langle \phi^E_n, \psi^E_n\rangle$ forms an embedding of E_n into E.

Also, we view each E_n as a subset of E, that is, we identify $\phi^E_n(x)$ with x, where $x \in E_n$; and for $e \in E$, $\psi^E_n(e) = e_n \in E_n$. Thus, $E = \bigcup_{n\in\mathbb{N}} E_n$.

To illustrate our construction, we generate the three first approximations of E, i.e., E_0, E_1 and E_2, and we look into the corresponding embeddings.

As explained above, E_0 is the one-point domain represented by $\{\perp_{E_0}\}$, and $E_1 = \{\perp_{E_1}, e_1\}$, where $e_1(a_0) = a_0$ and $\{a_0\} = A_0 = [C \to E_0]$ is such that $a_0(S^n(\perp)) = \perp_{E_0}$ for all $n \in \mathbb{N}^\infty$. Now $E_2 = [A_1 \to A_1]_\perp$, where $A_1 = [C \to E_1]$ (see Figure 2). The elements of A_1 are functions $a_{1,n}$ ($n \in \mathbb{N}$)

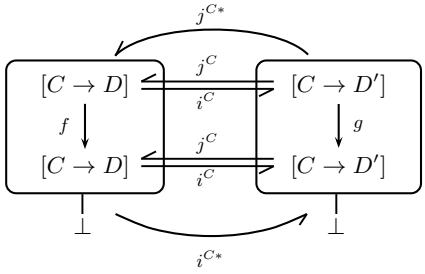


Figure 1: Embeddings on resourced domains

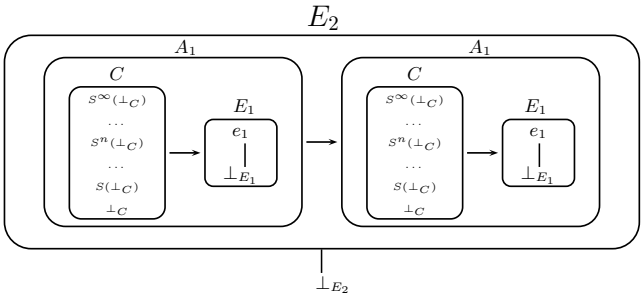


Figure 2: The domain E_2

such that:

$$a_{1,n}(S^k(\bot)) = \begin{cases} \bot_{E_1} & \text{if } k < n \\ e_1 & \text{if } k \geq n \end{cases}$$

We define $a_{1,\infty}$ as the function verifying that forall $n \in \mathbb{N}^\infty$ $a_{1,\infty}(S^n(\bot)) = \bot_{E_1}$, i.e., this is the least defined function in A_1. Notice that the most defined one is $a_{1,0}$, the constant function with $a_{1,0}(S^n(\bot)) = e_1$ forall $n \in \mathbb{N}^\infty$. Hence, we have an ordering in A_1 where $a_{1,\infty} \sqsubseteq a_{1,m} \sqsubseteq a_{1,n} \sqsubseteq a_{1,0}$ for all $m, n \in \mathbb{N}$ such that $n \leq m$.

Since $A_1 = \{a_{1,n} \mid n \in \mathbb{N}^\infty\}$, any increasing ω-chain in A_1 has as l.u.b. $a_{1,k}$ for some $k \in \mathbb{N}$, except for the constant chain $\{x_n = a_{1,\infty}\}_{n\in\mathbb{N}}$. Hence, we can characterize the continuous functions $e : A_1 \to A_1$ as those that satisfy the following property:

$$n \leq m \Rightarrow l \leq k, \text{ for } e(a_{1,m}) = a_{1,k} \text{ and } e(a_{1,n}) = a_{1,l} \quad (1)$$

Summarizing,

$$E_2 = \{\bot_{E_2}\} \cup \{e : \{a_{1,n}\}_{n\in\mathbb{N}^\infty} \to \{a_{1,n}\}_{n\in\mathbb{N}^\infty} \mid e \text{ sat.}(1)\}.$$

The representation of the embeddings $\langle i_0, j_0\rangle$ of E_0 into E_1 and $\langle i_1, j_1\rangle$ of E_1 into E_2 is shown in Figure 3. For E_2 we only highlight the constant functions $e_{2,\infty}$ and $e_{2,0}$ representing the two extremes of $[A_1 \to A_1]$, i.e., for all $n \in \mathbb{N}^\infty$

$$e_{2,\infty}(a_{1,n}) = a_{1,\infty} \quad \text{and} \quad e_{2,0}(a_{1,n}) = a_{1,0}.$$

2.3 Application operations

Application operations are defined in each domain E_n as $\mathsf{Ap}^\bot_{E_n} : E_{n+1} \times A_n \times C \to E_n$,

$$\mathsf{Ap}^\bot_{E_n}(e_{n+1}, a_n, c) \stackrel{\text{def}}{=} \begin{cases} \bot_{E_n} & \text{if } e_{n+1} = \bot_{E_{n+1}} \\ e_{n+1}(a_n)(c) & \text{if } e_{n+1} \neq \bot_{E_{n+1}} \end{cases}$$

ψ^E_n denotes the n-projection defined in Section 2.2 for the domain E. From the embedding $\langle \phi^E_n, \psi^E_n\rangle$ of E_n into E and using Corollary 2, we obtain the embedding $\langle (\phi^E_n)^C, (\psi^E_n)^C\rangle$ of $[C \to E_n]$ into $[C \to E]$, so that $\psi^{[C\to E]}_n$ stands for $(\psi^E_n)^C$. Application operation in E is $\mathsf{AP}^\bot_E : E \times [C \to E] \times C \to E$,

$$\mathsf{AP}^\bot_E(e, a, c) \stackrel{\text{def}}{=} \bigsqcup_{n\in\mathbb{N}} \mathsf{Ap}^\bot_{E_n}(\psi^E_{n+1}(e), \psi^{[C\to E]}_n(a), c).$$

In the following, we use $e\,(a)\,(c)$ for both $\mathsf{Ap}^\bot_{E_n}(e, a, c)$ and $\mathsf{AP}^\bot_E(e, a, c)$; from the context it should be clear which is the correct one. Notice that

$$\psi^E_n(e(a)(c)) = \psi^E_{n+1}(e)(\psi^{[C\to E]}_n(a))(c). \quad (2)$$

Analogously, from the definition of the application operation in the standard lifted function space D described in [2], where D is the initial solution of $D = [D \to D]_\bot$ and this initial solution can be described as an inverse limit similarly to E, it can be observed that

$$\psi^D_n(d(d')) = \psi^D_{n+1}(d)(\psi^D_n(d')). \quad (3)$$

3. SIMILARITY

In this section we define a relation between the standard lifted function domain $D = [D \to D]_\bot$ and the resourced domain $E = [[C \to E] \to [C \to E]]_\bot$ constructed in the previous section. Our relation is inspired by the *applicative bisimulation* defined by Abramsky in [1], so that functions $d \in D$ and $e \in E$ are considered to be *similar* if d and e have a *similar* applicative behaviour when infinite resources are available for e, i.e., they produce *similar* values when applied to *similar* arguments.

To formalize this circular definition we resort to a layered construction, that is, we inductively define a sequence of relations $\{ \lhd\rhd_n \subseteq D_n \times E_n\}_{n\in\mathbb{N}}$.

DEFINITION 1 (n-SIMILARITY). *A family of n-similarity relations between elements of D_n and E_n, for $n \in \mathbb{N}$, is defined as follows:*

- *0-similarity, $\lhd\rhd_0 : \bot_{D_0} \lhd\rhd_0 \bot_{E_0}$.*
- *n+1-similarity, $\lhd\rhd_{n+1}$, is the least relation in $D_{n+1} \times E_{n+1}$ that verifies:*
  (1) $\bot_{D_{n+1}} \lhd\rhd_{n+1} \bot_{E_{n+1}}$ and,
  (2) let $d \in D_{n+1}$ and $e \in E_{n+1}$ such that $d \neq \bot_{D_{n+1}}$ and $e \neq \bot_{E_{n+1}}$, then $d \lhd\rhd_{n+1} e$ if for any $d' \in D_n$ and for any $a' \in A_n$, $d' \lhd\rhd_n a'(S^\infty) \Rightarrow d(d') \lhd\rhd_n e(a')(S^\infty)$.

For a better understanding of this definition, we show the three first layers (Figure 4):

• 0-*similarity* ($\lhd\rhd_0$): $D_0 = \{\bot_{D_0}\}$ and $E_0 = \{\bot_{E_0}\}$, so that the relation reduces to $\bot_{D_0} \lhd\rhd_0 \bot_{E_0}$.

• 1-*similarity* ($\lhd\rhd_1$): $D_1 = \{\bot_{D_1}, d_1\}$, where $d_1 \in [D_0 \to D_0]$ is the function mapping $\bot_{D_0}$ into $\bot_{D_0}$, and recall that $E_1 = \{\bot_{E_1}, e_1\}$ with $e_1(a_0) = a_0$. $\bot_{D_1} \lhd\rhd_1 \bot_{E_1}$, by definition. Now let $d' \in D_0$ and $a' \in A_0 = \{a_0 \mid \forall n \in \mathbb{N}^\infty.\ a_0(S^n(\bot)) = \bot_{E_0}\}$, it must be that $d' = \bot_{D_0}$ and $a' = a_0$. Hence, we have that $\bot_{D_0} = d' \lhd\rhd_0 a'(S^\infty) = \bot_{E_0}$. Moreover, $d_1(d') = \bot_{D_0} \lhd\rhd_0 \bot_{E_0} = a_0(S^\infty) = e_1(a_0)(S^\infty) = e_1(a')(S^\infty)$. Thus, $d_1 \lhd\rhd_1 e_1$.

• 2-*similarity* ($\lhd\rhd_2$): $D_2 = \{\bot_{D_2}, d_{2,1}, d_{2,2}, d_{2,3}\}$, where the functions $d_{2,1}, d_{2,2}$ and $d_{2,3}$ are defined as follows:

$$\begin{array}{lll} d_{2,1}(\bot_{D_1}) = \bot_{D_1} & d_{2,2}(\bot_{D_1}) = \bot_{D_1} & d_{2,3}(\bot_{D_1}) = d_1 \\ d_{2,1}(d_1) = \bot_{D_1} & d_{2,2}(d_1) = d_1 & d_{2,3}(d_1) = d_1 \end{array}$$

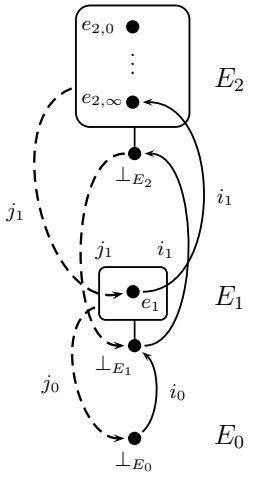


Figure 3: First steps of the construction of E

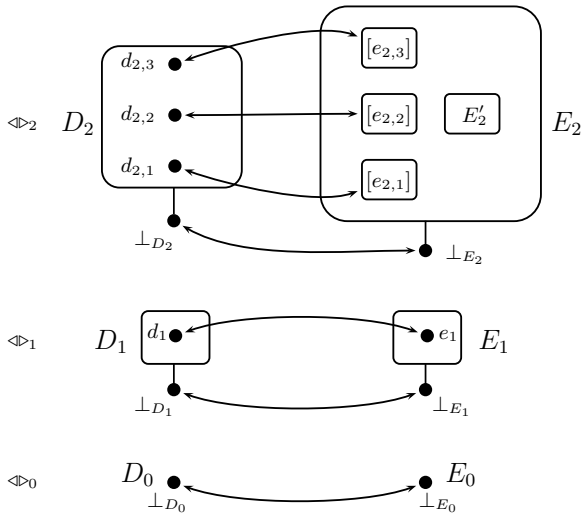


Figure 4: 0,1,2-similarity

We have shown in Subsection 2.2 that
$E_2 = \{\perp_{E_2}\} \cup \{e : \{a_{1,n}\}_{n\in\mathbb{N}^\infty} \to \{a_{1,n}\}_{n\in\mathbb{N}^\infty} \mid e \text{ sat. } (1)\}$.
By definition, $\perp_{D_2} \lhd\rhd_2 \perp_{E_2}$.
Let $d' \in D_1$ and $a' \in A_1 = [C \to E_1]$ such that $d' \lhd\rhd_1 a'(S^\infty)$. Consequently, either $d' = \perp_{D_1}$ and $a' = a_{1,\infty}$ (and hence $a'(S^\infty) = \perp_{E_1}$), or $d' = d_1$ and $a' = a_{1,k}$ for some $k \in \mathbb{N}$ (notice that $\forall n \in \mathbb{N} \,.\, a_{1,n}(S^\infty) = e_1$). Therefore, the functions in A_1 can be partitioned into two classes: $[a_{1,\infty}]$ is the class of functions returning $\perp_{E_1}$ when applied to S^∞; whereas the functions in $[a_{1,0}]$ return e_1 when infinite resources are provided, i.e., $[a_{1,0}] = \{a_{1,n} \mid a_{1,n}(S^\infty) = e_1\}$. Notice that the first class contains a unique element, that is, $[a_{1,\infty}] = \{a_{1,\infty}\}$. But it could be proved that 2-similarity requires that the classes in A_1 are preserved, i.e., for any $e \in E_2$ there exists $d \in D_2$ such that $d \lhd\rhd_2 e$ only if

$$\forall a, a' \in A_1.(a(S^\infty) = a'(S^\infty) \Rightarrow e(a)(S^\infty) = e(a')(S^\infty)).$$

Therefore, $E_2 = \{\perp_{E_2}\} \cup [e_{2,1}] \cup [e_{2,2}] \cup [e_{2,3}] \cup E_2'$. where

$$\begin{aligned}
[e_{2,1}] &= \{e \in E_2 \mid \forall a \in A_1 \,.\, e(a) \in [a_{1,\infty}]\},\\
[e_{2,2}] &= \{e \in E_2 \mid \forall a \in A_1 \,.\, (a \in [a_{1,\infty}] \Rightarrow e(a) \in [a_{1,\infty}])\\
&\qquad\qquad \wedge (a \in [a_{1,0}] \Rightarrow e(a) \in [a_{1,0}])\},\\
[e_{2,3}] &= \{e \in E_2 \mid \forall a \in A_1 \,.\, e(a) \in [a_{1,0}]\}.
\end{aligned}$$

Hence, we have that $\perp_{D_2} \lhd\rhd_2 \perp_{E_2}$, $\forall e \in [e_{2,1}].d_{2,1} \lhd\rhd_2 e$, $\forall e \in [e_{2,2}].d_{2,2} \lhd\rhd_2 e$, $\forall e \in [e_{2,3}].d_{2,3} \lhd\rhd_2 e$, and E_2' contains the elements of E_2 for which there is no similar element in D_2. This is graphically represented in Figure 4.

Next we prove some useful properties of the n-similarity, e.g., that it preserves undefinedness:

LEMMA 6. *Let $n \in \mathbb{N}$, $d \in D_n$ and $e \in E_n$. If $d \lhd\rhd_n e$ then either $(d = \perp_{D_n} \wedge e = \perp_{E_n})$ or $(d \neq \perp_{D_n} \wedge e \neq \perp_{E_n})$.*

PROOF. Trivial by the definition of $\lhd\rhd_n$. □

In the following lemma $\langle i_n^D, j_n^D\rangle$ stands for the embedding of D_n into D_{n+1} [1], $\langle i_n^E, j_n^E\rangle$ for the embedding of E_n into E_{n+1}, and $\langle i_n^C, j_n^C\rangle$ for the embedding of $[C \to E_n]$ into $[C \to E_{n+1}]$. The lemma states that the injections and projections of these embeddings preserve the similarity relation.

LEMMA 7. *Let $n \in \mathbb{N}$, $d \in D_{n+1}$, $e \in E_{n+1}$, $a \in A_{n+1}$, $d' \in D_n$, $e' \in E_n$ and $a' \in A_n$:*

1. $d \lhd\rhd_{n+1} e \Rightarrow j_n^D(d) \lhd\rhd_n j_n^E(e)$,
2. $d' \lhd\rhd_n e' \Rightarrow i_n^D(d') \lhd\rhd_{n+1} i_n^E(e')$,
3. $d \lhd\rhd_{n+1} a(S^\infty) \Rightarrow j_n^D(d) \lhd\rhd_n j_n^C(a)(S^\infty)$, *and*
4. $d' \lhd\rhd_n a'(S^\infty) \Rightarrow i_n^D(d') \lhd\rhd_{n+1} i_n^C(a')(S^\infty)$.

PROOF. By induction on n.

• $n = 0$: By definition, $j_0^D(d) = \perp_{D_0}$ for any $d \in D_1$, $j_0^E(e) = \perp_{E_1}$ for any $e \in E_1$, and $j_0^C(a)(S^\infty) = \perp_{E_0}$ for any $a \in A_1$. Therefore, $j_0^D(d) \lhd\rhd_0 j_0^E(e)$ and $j_0^D(d) \lhd\rhd_0 j_0^E(a)(S^\infty)$.

Likewise, $i_0^D(d') = \perp_{D_1}$ for any $d' \in D_0$, $i_0^E(e') = \perp_{E_1}$ for any $e' \in E_0$, and $i_0^C(a')(S^\infty) = \perp_{E_1}$ for any $a' \in A_0$. Therefore, $i_0^D(d') \lhd\rhd_1 i_0^E(e')$ and $i_0^D(d') \lhd\rhd_1 i_0^C(a')(S^\infty)$.

• $n > 0$:

(1). We assume $d \lhd\rhd_{n+1} e$, hence (by Lemma 6) either $d = \perp_{D_{n+1}}$ and $e = \perp_{E_{n+1}}$, or $d \neq \perp_{D_{n+1}}$ and $e \neq \perp_{E_{n+1}}$.

The first case is trivial by the definiton of j_n^D and j_n^E.

To prove the second case, let $d'' \in D_{n-1}$, and $a'' \in A_{n-1}$ such that $d'' \lhd\rhd_{n-1} a''(S^\infty)$. By induction hypothesis (4), we have that $i_{n-1}^D(d'') \lhd\rhd_n i_{n-1}^C(a'')(S^\infty)$. As $d \lhd\rhd_{n+1} e$, it must be that $d(i_{n-1}^D(d'')) \lhd\rhd_n e(i_{n-1}^C(a''))(S^\infty)$.
Then, by induction hypothesis (3),

$$j_{n-1}^D(d(i_{n-1}^D(d''))) \lhd\rhd_{n-1} j_{n-1}^C(e(i_{n-1}^C(a')))(S^\infty),$$

and by definition of j_n^D and j_n^E we have that

$$j_n^D(d)(d'') \lhd\rhd_{n-1} j_n^E(e)(a'')(S^\infty).$$

Therefore we have proved that $j_n^D(d) \lhd\rhd_n j_n^E(e)$.

(4). We assume $d' \lhd\rhd_n a'(S^\infty)$, thus (by Lemma 6) either $d' = \perp_{D_n}$ and $a'(S^\infty) = \perp_{E_n}$, or $d' \neq \perp_{D_n}$ and $a'(S^\infty) \neq \perp_{E_n}$.

The first case is trivial by the definition of i_n^D and i_n^C.

For the second case, let $d'' \in D_n$ and $a'' \in A_n$ such that $d'' \lhd\rhd_n a''(S^\infty)$. By induction hypothesis (3),

$$j_{n-1}^D(d'') \lhd\rhd_{n-1} j_{n-1}^C(a'')(S^\infty).$$

As $d' \lhd\rhd_n a'(S^\infty)$, it must be that

$$d'(j_{n-1}^D(d'')) \lhd\rhd_{n-1} a'(S^\infty)(j_{n-1}^C(a''))(S^\infty).$$

Then, by induction hypothesis (4), we have that

$$i_{n-1}^D(d'(j_{n-1}^D(d''))) \lhd\rhd_n i_{n-1}^C(a'(S^\infty)(j_{n-1}^C(a')))(S^\infty),$$

and by definition of i_n^D and i_n^C we have that

$$i_n^D(d)(d'') \lhd\rhd_n i_n^C(a'(S^\infty))(a'')(S^\infty).$$

Therefore we have proved that $i_n^D(d') \lhd\rhd_{n+1} i_n^C(a')(S^\infty)$.
Proofs for (2) and (3) are similar to those for (1) and (4). □

The previous lemma enables to pass the similarity relation up and down through the approximations of D and E. We define a similarity relation between functions in D and E.

DEFINITION 2 (SIMILARITY). *$\lhd\rhd$ is defined as the least relation in $D \times E$ that verifies that for each $d \in D$ and $e \in E$, $d \lhd\rhd e$ if $\forall n \in \mathbb{N}$. $\psi_n^D(d) \lhd\rhd_n \psi_n^E(e)$.*

Similarity preserves undefinedness.

COROLLARY 8. *Let $d \in D$ and $e \in E$. If $d \lhd\rhd e$ then either $(d = \perp_D \wedge e = \perp_E)$ or $(d \neq \perp_D \wedge e \neq \perp_E)$.*

PROOF. Is a corollary of Lemma 6. □

We give an alternative characterization for $\lhd\rhd$ which is more convenient for writing proofs involving $\lhd\rhd$.

PROPOSITION 9. *Let $d \in D$, $e \in E$. $d \lhd\rhd e$ if and only if:*

- $(d = \perp_D \wedge e = \perp_E)$, *or*
- $(d \neq \perp_D \wedge e \neq \perp_E) \wedge \forall d' \in D. \forall a' \in [C \to E].$
  $d' \lhd\rhd a'(S^\infty) \Rightarrow d(d') \lhd\rhd e(a')(S^\infty)$.

PROOF. To simplify the notation we write r_n for $\psi_n^R(r)$ where $n \in \mathbb{N}$, $r \in R$ and $R \in \{D, E, [C \to E]\}$.
(if) We first prove that if d and e verify the property then they are similar.

- *Case 1:* $d = \perp_D$ and $e = \perp_E$.

We have that $(\perp_D)_n = \perp_{D_n}$ and $(\perp_E)_n = \perp_{E_n}$ for all $n \in \mathbb{N}$. By definiton of $\lhd\rhd_n$, $(\perp_D)_n = \perp_{D_n} \lhd\rhd_n \perp_{E_n} = (\perp_E)_n$, for any $n \in \mathbb{N}$. Therefore, $d \lhd\rhd e$.

- *Case 2:* $d \neq \perp_D$ and $e \neq \perp_E$.

By assumption, $d'' \lhd\rhd a''(S^\infty) \Rightarrow d(d'') \lhd\rhd e(a'')(S^\infty)$, for any $d'' \in D$ and any $a'' \in [C \to E]$.
By definition of $\lhd\rhd$, we can assure that if $d(d'') \lhd\rhd e(a'')(S^\infty)$, then $\forall m \in \mathbb{N}.(d(d''))_m \lhd\rhd_m (e(a'')(S^\infty))_m$. Then, by the equations (2) and (3) in Section 2.3, we have that
$\forall m \in \mathbb{N}. d_{m+1}(d''_m) \lhd\rhd_m e_{m+1}(a''_m)(S^\infty)$.
Consequently, for any $d'' \in D$ and any $a'' \in [C \to E]$,

$$d'' \lhd\rhd a''(S^\infty) \Rightarrow \forall m \in \mathbb{N}. d_{m+1}(d'_m) \lhd\rhd_m e_{m+1}(a'_m)(S^\infty). \quad (4)$$

Now we have to prove that $d \lhd\rhd e$, i.e., $\forall n \in \mathbb{N}. d_n \lhd\rhd_n e_n$.
But $d_0 = \perp_{D_0}$ and $e_0 = \perp_{E_0}$ and, by definition, $\perp_{D_0} \lhd\rhd_0 \perp_{E_0}$.
Thus, we have to check that $\forall n > 0. d_n \lhd\rhd_n e_n$, or equivalently, that $\forall n \in \mathbb{N}. d_{n+1} \lhd\rhd_{n+1} e_{n+1}$.
That is, it must be verified that for any $d' \in D_n$ and for any $a' \in A_n$ $d' \lhd\rhd_n a'(S^\infty) \Rightarrow d_{n+1}(d') \lhd\rhd_n e_{n+1}(a')(S^\infty)$.
Let $d' \in D_n$, we construct a tuple

$$\overline{d'} = \langle d'_0, d'_1, \ldots, d'_{n-1}, d', d'_{n+1} \ldots \rangle$$

such that $d'_k = j_k^D(d'_{k+1})$ for $k < n$ and $d'_k = i_{k-1}(d'_{k-1})$ for $k > n$. Therefore, $\overline{d'} \in D$.
Likewise, let $a' \in A_n = [C \to E_n]$, we construct

$$\overline{a'} = \langle a'_0, a'_1, \ldots, a'_{n-1}, a', a_{n+1} \ldots \rangle \in [C \to E].$$

By Lemma 7 we can assure that $\overline{d'} \lhd\rhd \overline{a'}(S^\infty)$ whenever $d' \lhd\rhd_n a'(S^\infty)$. Then, by (4) we have that

$$\forall m \in \mathbb{N}. d_{m+1}(\overline{d'}_m) \lhd\rhd_m e_{m+1}(\overline{a'}_m)(S^\infty),$$

and particularly: $d_{n+1}(d') \lhd\rhd_n e_{n+1}(a')(S^\infty)$.
(only if) Let us prove that if $d \lhd\rhd e$ then d and e satisfy the property. By Corollary 8 we only have two cases:

- *Case 1:* $d = \perp_D$ and $e = \perp_E$. Trivial.
- *Case 2:* $d \neq \perp_D$ and $e \neq \perp_E$.

Let $d' \in D$ and $a' \in [C \to E]$ such that $d' \lhd\rhd a'(S^\infty)$.
We have to prove that $d(d') \lhd\rhd e(a')(S^\infty)$, that is,
$\forall n \in \mathbb{N}.(d(d'))_n \lhd\rhd_n (e(a')(S^\infty))_n$.
For $n = 0$ this is trivial. Now consider $n > 0$; by hypothesis, $d \lhd\rhd e$ and $d' \lhd\rhd a'(S^\infty)$, therefore, by definition of $\lhd\rhd$ we have that $\forall m \in \mathbb{N}. d_m \lhd\rhd_m e_m \wedge d'_m \lhd\rhd_m a'_m(S^\infty)$.
Particularly: $d_{n+1} \lhd\rhd_{n+1} e_{n+1}$ and $d'_n \lhd\rhd_n a'_n(S^\infty)$.
Consequently, $d_{n+1}(d'_n) \lhd\rhd_n e_{n+1}(a'_n)(S^\infty)$, and therefore, $d(d')_n \lhd\rhd_n (e(a')(S^\infty))_n$, by (2) and (3) in Section 2.3. □

4. A DENOTATIONAL SEMANTICS FOR A LAZY λ-CALCULUS

The language described in [7] is a normalised λ-calculus extended with recursive *lets*. The restricted syntax is given in Figure 5, where all bound variables are distinct, and applications are of an expression to a variable.

We consider a *heap* Γ as a finite partial function from variables to expressions, that is, $Heap \in \mathcal{P}_f(Var \times Exp)$, with all the variables different pairwise.

4.1 A standard denotational semantics

To define a denotational semantics for this calculus, a domain of values and environments to associate values to the variables are needed.

An environment maps variables into values,

$$\rho \in Env = Var \to Value,$$

where *Value* is some appropriate domain containing at least a lifted version of its own function space, i.e.,

$$v \in Value = [Value \to Value]_\perp.$$

Notice that *Value* corresponds to the standard lifted domain D described in [2].

An ordering is defined on environments, such that if ρ is less or equal than ρ' then ρ' may bind more variables than ρ, but otherwise is equal to ρ. Formally, let $\rho, \rho' \in Env$ be environments, ρ is *less or equal* to ρ' (denoted as $\rho \leq \rho'$) iff $\forall x. \rho\ x \neq \perp \Rightarrow \rho\ x = \rho'\ x$.

For the language described by the syntax in Figure 5, Launchbury defines a denotational semantics in [7]. The semantic function is $[\![-]\!] : Exp \to Env \to Value$, which is given in Figure 6.

The function $\{\!\{ - \}\!\} : Heap \to Env \to Env$ should be thought as an environment modifier defined as follows:

$$\{\!\{(x_i \mapsto e_i)_{i=1}^n\}\!\}\rho = \mu\rho'.\rho \sqcup (x_i \mapsto [\![e_i]\!]_{\rho'})_{i=1}^n,$$

where μ stands for the least fixed point operator and $\sqcup$ for

$$(\rho \sqcup (x \mapsto [\![e]\!]_{\rho'}))\ y = \begin{cases} \rho(y) & \text{if } y \neq x \\ [\![e]\!]_{\rho'} & \text{if } y = x \end{cases}$$

This definition only makes sense on environments ρ which are *consistent* with a heap Γ (i.e., if ρ and Γ bind the same variables, then they are bound to values for which an upper bound exists). This consistency is guaranteed in the denotational semantics definition in Figure 6 because we require that all bound variables are distinct.

$$
\begin{array}{rcll}
x & \in & \mathit{Var} & \\
e & \in & \mathit{Exp} & ::= \quad x \\
 & & & \mid \quad \lambda x.e \\
 & & & \mid \quad e\ x \\
 & & & \mid \quad \mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e
\end{array}
$$

Figure 5: Syntax

$$
\begin{array}{rl}
[\![x]\!]_\rho & = \rho\ x \\
[\![\lambda x.e]\!]_\rho & = \mathsf{up}(\lambda\nu.[\![e]\!]_{\rho \sqcup \{x \mapsto \nu\}}) \\
[\![e\ x]\!]_\rho & = \mathsf{AP}_D^\perp\ ([\![e]\!]_\rho, [\![x]\!]_\rho) \\
[\![\mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e]\!]_\rho & = [\![e]\!]_{\{\!(x_i \mapsto e_i)_{i=1}^{n}\!\}\rho}
\end{array}
$$

Figure 6: Denotational Semantics

4.2 A resourced denotational semantics

A resourced denotational semantics is also defined in [7], where the meaning of an expression depends on the number of available resources. For this a new domain of values, and the corresponding environments, are needed.

A *resourced environment* is a function mapping variables into functions from resources to values,

$$\sigma \in \mathit{CEnv} = \mathit{Var} \to [C \to \mathit{CValue}],$$

where CValue is some appropiate resourced domain, i.e.,

$$\mathit{CValue} = [[C \to \mathit{CValue}] \to [C \to \mathit{CValue}]]_\perp.$$

Notice that CValue corresponds to the resourced domain E constructed in Section 2.

An ordering is defined on resourced environments too, such that if σ is less or equal than σ' then, for any number of available resources, σ' may bind more variables than σ, but otherwise is equal to σ: Let $\sigma, \sigma' \in \mathit{CEnv}$ be resourced environments, σ is *less or equal* than σ' (denoted as $\sigma \leq \sigma'$) iff $\forall x \in \mathit{Var}$ and $\forall m \in \mathbb{N}$

$$\sigma x(S^m(\perp)) \neq \perp \Rightarrow \sigma\ x(S^m(\perp)) = \sigma' x(S^m(\perp)).$$

The resourced semantics focuses on approximations to the semantics of Figure 6. The semantic function $\mathcal{N}[\![-]\!] : \mathit{Exp} \to \mathit{CEnv} \to [C \to \mathit{CValue}]$ is given in Figure 7.

The function $\mathcal{N}\{\!\{-\}\!\} : \mathit{Heap} \to \mathit{CEnv} \to \mathit{CEnv}$ is defined analogously to $\{\!\{-\}\!\}$:

$$\mathcal{N}\{\!\{(x_i \mapsto E_i)_{i=1}^{n}\}\!\}\sigma = \mu\sigma'.\sigma \sqcup (x_i \mapsto \mathcal{N}[\![E_i]\!]_{\sigma'})_{i=1}^{n}.$$

4.3 Equivalence

In Section 3 we have shown how to relate values in Value with values in CValue, but we need to extend the notion of similarity to environments:

Definition 3. (*Similarity of environments*). *Let $\rho \in \mathit{Env}$ be an environment and $\sigma \in \mathit{CEnv}$ be a resourced environment. ρ and σ are* similar *(denoted by $\rho \lhd\rhd \sigma$) when*

$$\forall x \in \mathit{Var}.\rho\ x \lhd\rhd \sigma\ x(S^\infty).$$

Now we can prove the equivalence between the standard denotational semantics and the resourced one. More precisely, we prove that they produce *similar* values.

Theorem 10. *Let $e \in \mathit{Exp}$ be an expression and $\rho \in \mathit{Env}$, $\sigma \in \mathit{CEnv}$ be similar environments (i.e., $\rho \lhd\rhd \sigma$), then*

$$[\![e]\!]_\rho \lhd\rhd \mathcal{N}[\![e]\!]_\sigma\ (S^\infty).$$

Proof. By structural induction on e:

$\mathbf{e \equiv x}$

By definition, $[\![x]\!]_\rho = \rho\ x$ and $\mathcal{N}[\![x]\!]_\sigma\ (S^\infty) = \sigma\ x(S^\infty)$.
By hypothesis $\rho \lhd\rhd \sigma$, therefore:
$[\![x]\!]_\rho = \rho\ x \lhd\rhd \sigma\ x(S^\infty) = \mathcal{N}[\![x]\!]_\sigma\ (S^\infty)$.

$\mathbf{e \equiv \lambda x.e'}$

By definition, $[\![\lambda x.e']\!]_\rho = \mathsf{up}(\lambda\nu.[\![e']\!]_{\rho \sqcup \{x \mapsto \nu\}}) \neq \perp_{\mathit{Value}}$ and
$\mathcal{N}[\![\lambda x.e']\!]_\sigma\ (S^\infty) = \mathsf{up}(\lambda\tau.\mathcal{N}[\![e']\!]_{\sigma \sqcup \{x \mapsto \tau\}}\ ) \neq \perp_{\mathit{CValue}}$.
We have to prove that
$\mathsf{up}(\lambda\nu.[\![e']\!]_{\rho \sqcup \{x \mapsto \nu\}}) \lhd\rhd \mathsf{up}(\lambda\tau.\mathcal{N}[\![e']\!]_{\sigma \sqcup \{x \mapsto \tau\}}\ )$.
We use the alternative characterization of similarity (Proposition 9). Let $v \in \mathit{Value}$ and $f \in C \to \mathit{CValue}$ such that $v \lhd\rhd f(S^\infty)$, then:
$\mathsf{AP}_D^\perp\ (\mathsf{up}(\lambda\nu.[\![e']\!]_{\rho \sqcup \{x \mapsto \nu\}}), v) = [\![e']\!]_{\rho \sqcup \{x \mapsto v\}}$ and
$\mathsf{AP}_E^\perp\ (\mathsf{up}(\lambda\tau.\mathcal{N}[\![e']\!]_{\sigma \sqcup \{x \mapsto \tau\}}\ ), f, S^\infty) = \mathcal{N}[\![e']\!]_{\sigma \sqcup \{x \mapsto f\}}\ (S^\infty)$.
If $\rho' = \rho \sqcup \{x \mapsto v\} \lhd\rhd \sigma \sqcup \{x \mapsto f\} = \sigma'$, then, by induction hypothesis, we get the desired result.
Let us prove that $\rho' \lhd\rhd \sigma'$:

- If $y \neq x$ then $\rho'\ y = \rho\ y \lhd\rhd \sigma\ y(S^\infty) = \sigma'\ y$, because $\rho \lhd\rhd \sigma$.
- If $y = x$ then $\rho'\ y = v \lhd\rhd f(S^\infty) = \sigma'\ y$, by hypothesis.

Thus, $[\![e']\!]_{\rho'} \lhd\rhd \mathcal{N}[\![e']\!]_{\sigma'}\ (S^\infty)$.
Therefore, we have proved that $[\![\lambda x.e']\!]_\rho \lhd\rhd \mathcal{N}[\![\lambda x.e']\!]_\sigma\ (S^\infty)$.

$\mathbf{e \equiv e'\ x}$

By definition, $[\![e'\ x]\!]_\rho = \mathsf{AP}_D^\perp\ ([\![e']\!]_\rho, [\![x]\!]_\rho)$ and
$\mathcal{N}[\![e'\ x]\!]_\sigma\ (S^\infty) = \mathsf{AP}_E^\perp\ (\mathcal{N}[\![e']\!]_\sigma\ S^\infty, \mathcal{N}[\![x]\!]_\sigma\ , S^\infty)$.
Thus, we have to prove that
$\mathsf{AP}_D^\perp\ ([\![e']\!]_\rho, [\![x]\!]_\rho) \lhd\rhd \mathsf{AP}_E^\perp\ (\mathcal{N}[\![e']\!]_\sigma\ (S^\infty), \mathcal{N}[\![x]\!]_\sigma\ , S^\infty)$.
By induction hypothesis, $[\![e']\!]_\rho \lhd\rhd \mathcal{N}[\![e']\!]_\sigma\ (S^\infty)$. Then by Corollary 8 there are two cases:

- *Case 1:* $[\![e']\!]_\rho = \perp_{\mathit{Value}}$ and $\mathcal{N}[\![e']\!]_\sigma\ (S^\infty) = \perp_{\mathit{CValue}}$.
  $[\![e'\ x]\!]_\rho = \mathsf{AP}_D^\perp\ ([\![e']\!]_\rho, [\![x]\!]_\rho) = \mathsf{AP}_D^\perp\ (\perp_{\mathit{Value}}, [\![x]\!]_\rho) = \perp_{\mathit{Value}}$
  $\lhd\rhd \perp_{\mathit{CValue}} = \mathsf{AP}_E^\perp\ (\perp_{\mathit{CValue}}, \mathcal{N}[\![x]\!]_\sigma\ , S^\infty)$
  $= \mathsf{AP}_E^\perp\ (\mathcal{N}[\![e']\!]_\sigma\ (S^\infty), \mathcal{N}[\![x]\!]_\sigma\ , S^\infty) = \mathcal{N}[\![e'\ x]\!]_\sigma\ (S^\infty)$.
- *Case 2:* $\perp_{\mathit{Value}} \neq [\![e']\!]_\rho \lhd\rhd \mathcal{N}[\![e']\!]_\sigma\ (S^\infty) \neq \perp_{\mathit{CValue}}$.
  Thus, $\mathsf{AP}_D^\perp\ ([\![e']\!]_\rho, [\![x]\!]_\rho) = [\![e']\!]_\rho([\![x]\!]_\rho)$ and
  $\mathsf{AP}_E^\perp\ (\mathcal{N}[\![e']\!]_\sigma\ (S^\infty), \mathcal{N}[\![x]\!]_\sigma\ , S^\infty) =$
  $\mathcal{N}[\![e']\!]_\sigma\ (S^\infty)(\mathcal{N}[\![x]\!]_\sigma)\ (S^\infty)$.
  By induction hypothesis $[\![x]\!]_\rho \lhd\rhd \mathcal{N}[\![x]\!]_\sigma\ (S^\infty)$.
  Therefore, by the alternative characterization for $\lhd\rhd$:
  $[\![e']\!]_\rho([\![x]\!]_\rho) \lhd\rhd \mathcal{N}[\![e']\!]_\sigma\ (S^\infty)(\mathcal{N}[\![x]\!]_\sigma)\ (S^\infty)$.
  So that $[\![e'\ x]\!]_\rho \lhd\rhd \mathcal{N}[\![e'x]\!]_\sigma\ (S^\infty)$.

$\mathbf{e \equiv \mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e'}$

By definition:
$[\![\mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e']\!]_\rho = [\![e']\!]_{\{\!(x_i \mapsto e_i)_{i=1}^{n}\!\}\rho}$ and
$\mathcal{N}[\![\mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e']\!]_\sigma\ (S^\infty) =$
$\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!(x_i \mapsto e_i)_{i=1}^{n}\!\}\sigma}\ (S^\infty)$.
Using fix point techniques it can be proved that if $\rho \lhd\rhd \sigma$ then $\{\!\{\Gamma\}\!\}\rho \lhd\rhd \mathcal{N}\{\!\{\Gamma\}\!\}\sigma$ for any heap Γ consistent with ρ and σ. Thus, $\{\!\{(x_i \mapsto e_i)_{i=1}^{n}\}\!\}\rho \lhd\rhd \mathcal{N}\{\!\{(x_i \mapsto e_i)_{i=1}^{n}\}\!\}\sigma$.
By induction hypothesis:
$[\![e']\!]_{\{\!(x_i \mapsto e_i)_{i=1}^{n}\!\}\rho} \lhd\rhd \mathcal{N}[\![e']\!]_{\mathcal{N}\{\!(x_i \mapsto e_i)_{i=1}^{n}\!\}\sigma}\ (S^\infty)$.
Therefore, we obtain that
$[\![\mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e']\!]_\rho \lhd\rhd$
$\mathcal{N}[\![\mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e']\!]_\sigma\ (S^\infty)$. □

This result allows to prove the computational adequacy of the natural semantics with respect to the denotational one.

$$
\begin{array}{rcl}
\mathcal{N}[\![e]\!]_\sigma\ (\bot) & = & \bot \\
\mathcal{N}[\![x]\!]_\sigma\ (S^{k+1}(\bot)) & = & \sigma\ x\ (S^k(\bot)) \\
\mathcal{N}[\![\lambda x.e]\!]_\sigma\ (S^{k+1}(\bot)) & = & \mathsf{up}(\lambda\tau.\mathcal{N}[\![e]\!]_{\sigma\sqcup\{x\mapsto\tau\}}\ ) \text{ where } \tau : C \to \mathit{CValue} \\
\mathcal{N}[\![e\ x]\!]_\sigma\ (S^{k+1}(\bot)) & = & \mathsf{AP}^\bot_E\ (\mathcal{N}[\![e]\!]_\sigma\ (S^k(\bot)), \mathcal{N}[\![x]\!]_\sigma\ , S^k(\bot)) \\
\mathcal{N}[\![\mathtt{let}\ \{x_i = e_i\}_{i=1}^n\ \mathtt{in}\ e]\!]_\sigma\ (S^{k+1}(\bot)) & = & \mathcal{N}[\![e]\!]_{\mathcal{N}[\![(x_i\mapsto e_i)_{i=1}^n]\!]\sigma}\ (S^k(\bot))
\end{array}
$$

Figure 7: Resourced Denotational Semantics

5. CONCLUSIONS AND FUTURE WORK

We have tackled the problem of constructing a semantic domain for representing denotationally a resourced semantics. Abramsky and Ong defined in [2] an initial solution for the domain equation $D = [D \to D]_\bot$ suitable for defining a denotational semantics for a lazy λ-calculus. Following their schema, we have constructed the initial solution of the domain equation $E = [[C \to E] \to [C \to E]]_\bot$. We define a set of constructions on embeddings (adding a resource, lifting and function space), which can be cleanly defined by using function sections, and they simplify the presentation of the construction of E with respect to that of D in [2].

A resourced function space like E can be used to define a resourced denotational semantics that models the consumption of syntactic levels and thus, be nearer to syntax-oriented operational semantics based on rules, such as Launchbury's natural semantics for lazy evaluation. In fact, this is the approach taken by Launchbury in [7] in order to prove the computational adequacy of his natural semantics. But the equivalence between the standard and the resourced denotational semantics turns out to be not so easy to establish, because the semantic domains are different. The same problem can be found in [15].

To prove the computational adequacy it is only required that both semantics converge for the same expressions. However, in order to prove this result by structural induction a stronger property is needed, namely that both semantics produce values that behave "similarly". The problem arises in the application rule, because it depends on the semantics of the functional abstraction. Therefore, we have defined a similarity relation between the values of $D = [D \to D]_\bot$ and those of $E = [[C \to E] \to [C \to E]]_\bot$. Since a direct definition of the relation between D and E is not possible, we have first defined the similarity gradually between the approximation domains D_n and E_n. Afterwards, we prove that this layered definition satisfies an applicative bisimulation-like property.

We are interested in studying more properties of the similarity relation, especially in the context of category theory. We also want to compare it with the notions of bisimulation and bisimilarity —particularly with Abramsky's applicative bisimulation [1]— and with observational or contextual equivalences, like those in [10] or [12].

6. ACKNOWLEDGMENTS

We are thankful to John Launchbury for his valuable comments, and appreciate very much the help of David de Frutos Escrig. This work is partially supported by the grants: TIN2009-14599-C03-01, BES-2007-16823, S2009/TIC-1465.

7. REFERENCES

[1] S. Abramsky. *Research Topics in Functional Programming*, chapter The Lazy Lambda Calculus, pages 65–116. Addison-Wesley, 1990.

[2] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993.

[3] C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM International Conference on Functional Programming (ICFP'00)*, pages 162–173, 2000.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[5] G. Boudol, P. Curien, and C. Lavatelli. A semantics for lambda calculi with resources. *Mathematical Structures in Computer Science*, 9(4):437–482, 1999.

[6] C. Gunter and D. S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 633–674. Elsevier Science, 1990.

[7] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[8] A. Melton, D. A. Schmidt, and G. E. Strecker. Galois connections and computer science applications. In *Category Theory and Computer Programming*, pages 299–312. LNCS 240, Springer, 1986.

[9] K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.

[10] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Journal of Theoretical Computer Science*, 1(2):125–159, 1975.

[11] L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. *Trends in Functional Programming*, volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pages 65–80. Intellect, 2010.

[12] M. Schmidt-Schauğ. Equivalence of call-by-name and call-by-need for lambda-calculi with letrec. Technical report, Institut für Informatik. J. W. Goethe Universität Frankfurt am Main, Germany, 2006.

[13] D. S. Scott. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*, pages 97–136. LNCS 274, Springer, 1972.

[14] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[15] M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud University Nijmegen, The Netherlands, 2007.

A Locally Nameless Representation for a Natural Semantics for Lazy Evaluation

Lidia Sánchez-Gil[1], Mercedes Hidalgo-Herrero[2], and Yolanda Ortega-Mallén[1]

[1] Dpto. Sistemas Informáticos y Computación, Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain
[2] Dpto. Didáctica de las Matemáticas, Facultad de Educación, Universidad Complutense de Madrid, Spain

Abstract. We propose a locally nameless representation for Launchbury's natural semantics for lazy evaluation. Names are reserved for free variables, while bound variable names are replaced by indices. This avoids the use of α-conversion and Barendregt's variable convention, and facilitates proof formalization. Our definition includes the management of multi-binders to represent simultaneous recursive local declarations. We use cofinite quantification to express the semantic rules that require the introduction of fresh names, but we show that existential rules are admissible too. Moreover, we prove that the choice of names during the evaluation of a term is irrelevant as long as they are fresh enough.

1 Motivation

Call-by-need evaluation, which avoids repeated computations, is the semantic foundation for lazy functional programming languages like Haskell or Clean. Launchbury defines in [7] a natural semantics for lazy evaluation where the set of *bindings*, i.e., (variable, expression) pairs, is explicitly managed to make possible their sharing. In order to prove that this lazy semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury introduces some variations in his natural semantics. On the one hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach of this mechanism, applications are carried out in the alternative semantics by introducing indirections instead of by performing the β-reduction through substitution. On the other hand, the update of bindings with their computed values is an operational notion without counterpart in the standard denotational semantics, so that the alternative natural semantics does no longer update bindings and becomes a *call-by-name* semantics.

Unfortunately, the proof of the equivalence between the lazy natural semantics and its alternative version with indirections and nonupdate is detailed nowhere, and a simple induction turns out to be insufficient. Intuitively, both reduction systems should produce the same results. However, this cannot be directly established since final values may contain free variables which are dependent on the

A. Roychoudhury and M. D'Souza (Eds.): ICTAC 2012, LNCS 7521, pp. 105–119, 2012.

context of evaluation, which is represented by the heap of bindings. The changes introduced by the alternative semantics do deeply affect the heaps. Although indirections and "duplicated" bindings (a consequence of no updating) do not add relevant information to the context, it is awkward to prove this fact.

In the usual representation of the lambda-calculus, i.e., with variable names for free and bound variables, terms are identifed up to α-conversion. Dealing with α-equated terms usually implies the use of Barendregt's variable convention [3] to avoid the renaming of bound variables. However, the use of the variable convention in rule inductions is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [15]). Looking for a system of binding more amenable to formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [5]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [6], while free variables preserve their names. Hence, α-conversion is no longer needed and variable substitution is easily defined because there is no danger of name capture. Moreover, this representation is suitable for working with proof assistants like Coq [4] or Isabelle [9].

The present work is the first step to prove formally the equivalence between Launchbury's semantics and its alternative version. We start by defining a locally nameless representation of the λ-calculus extended with recursive local declarations. Then we express Launchbury's rules in the new style and present several properties of the reduction system that are useful for the equivalence proof.

Our concern for reproducing and formalizing the proof of this equivalence is not arbitrary. Launchbury's semantics has been cited frequently and has inspired many further works as well as several extensions [2,8,13,17], where the corresponding adequacy proofs have been obtained by just adapting Launchbury's proof scheme. We have extended ourselves the λ-calculus with a new expression that introduces parallelism when performing functional applications [11]. This *parallel application* creates new processes to distribute the computation; these processes exchange values through communication channels. The corresponding adequacy property relies on the adequacy of Launchbury's natural semantics.

The paper is structured as follows: In Section 2 we present the locally nameless representation of the lambda calculus extended with recursive local declarations. In Section 3 we describe a locally nameless translation of Launchbury's natural semantics for lazy evaluation [7], together with the corresponding regularity, introduction and renaming lemmas. The proofs (by hand) of these lemmas and other auxiliary results are detailed in [12]. In Section 4 we comment on some related work. The last two sections are devoted to conclusions and future work.

2 The Locally Nameless Representation

The language described by Launchbury in [7] is a normalized lambda calculus extended with recursive local declarations. We reproduce the restricted syntax in Figure 1. Normalization is achieved in two steps. First an α-conversion is carried out so that all bound variables have distinct names. In a second phase, arguments for applications are enforced to be variables. These static transformations simplify the definition of the reduction rules.

$$
\begin{array}{l}
x \in \mathit{Var} \\
e \in \mathit{Exp} ::= x \mid \lambda x.e \mid (e\ x) \mid \\
\qquad\qquad \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e
\end{array}
$$

Fig. 1. Restricted *named* syntax

$$
\begin{array}{lll}
x \in \mathit{Id} & & i, j \in \mathbb{N} \\
v \in \mathit{Var} & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in \mathit{LNExp} & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \\
 & & \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

Fig. 2. Locally nameless syntax

We give the corresponding locally nameless representation by following the methodology summarized in [5]:

1. Define the syntax of the extended λ-calculus in the locally nameless style.
2. Define the variable opening and variable closing operations.
3. Define the free variables and substitution functions, as well as the local closure predicate.
4. State and prove the properties of the operations on terms that are needed in the development to be carried out.

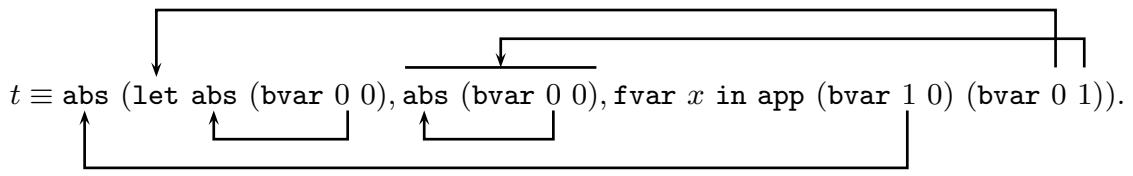
2.1 Locally Nameless Syntax

The locally nameless (restricted) syntax is shown in Figure 2. *Var* stands now for the set of *variables*, where *bound variables* and *free variables* are distinguished. The calculus includes two binding constructions: λ-abstraction and `let`-declaration. Being the latter a *multi-binder*, we follow Charguéraud [5] and represent bound variables with two natural numbers: The first number is a de Bruijn index that counts how many binders (abstraction or `let`) one needs to cross to the left to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind one variable; thus, the second number should be zero. In the following, a list like $\{t_i\}_{i=1}^{n}$ is represented as $\overline{t}$, with length $|\overline{t}| = n$.

Example 1. Let $e \in \mathit{Exp}$ an expression in the named representation:

$$e \equiv \lambda z.\texttt{let}\ x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2, x_3 = x\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in \mathit{LNExp}$ is:



Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms. □

As bound variables are nameless, the first phase of Launchbury's normalization is unneeded. However, application arguments are still restricted to variables.

$$
\begin{array}{lll}
\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) & = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases} \\
\{k \to \overline{x}\}(\texttt{fvar}\ x) & = \texttt{fvar}\ x \\
\{k \to \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \to \overline{x}\}\ t) \\
\{k \to \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \to \overline{x}\}\ t)\ (\{k \to \overline{x}\}\ v) \\
\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \to \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}\ t)
\end{array}
$$

where $\{k \to \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \to \overline{x}\}\,\cdot)\ \overline{t}$.

Fig. 3. Variable opening

2.2 Variable Opening and Variable Closing

Variable opening and *closing* are the main operations to manipulate locally nameless terms. We extend to `let` the definitions given by Charguéraud in [5].[1]

To explore the body of a binder (abstraction or `let`), one needs to replace the corresponding bound variables by fresh names. In the case of an abstraction `abs` t the *variable opening operation* replaces in t with a (fresh) name every bound variable which refers to the outermost abstraction. Analogously, to open `let` $\overline{t}$ `in` t we provide a list of $|\overline{t}|$ distinct fresh names to replace the bound variables that occur in $\overline{t}$ and in the body t which refer to this particular declaration.

Variable opening is defined by means of a more general function $\{k \to \overline{x}\}t$ (Figure 3), where the number k represents the nesting level of the binder to be opened, and $\overline{x}$ is a list of pairwise-distinct identifiers in Id. Since the level of the outermost binder is 0, variable opening is defined as: $t^{\overline{x}} = \{0 \to \overline{x}\}t$. We extend this operation to lists of terms: $\overline{t}^{\overline{x}} = \texttt{List.map}\ (\cdot^{\overline{x}})\ \overline{t}$.

The last definition and those in Figure 3 include some operations on lists. We use an ML-like notation. For instance, `List.nth` $j\ \overline{x}$ represents the $(j+1)^{th}$ element of $\overline{x}$,[2] and `List.map` $f\ \overline{t}$ indicates that the function f is applied to every term in the list $\overline{t}$. In the rest of definitions we will use similar list operations.

Example 2. Let $t \equiv \texttt{abs}\ (\texttt{let bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in app}\ (\texttt{abs bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1))$. Hence, the body of the abstraction is:

$$u \equiv \texttt{let bvar}\ 0\ 1, \boxed{\texttt{bvar}\ 1\ 0}\ \texttt{in app}\ (\texttt{abs}\ \boxed{\texttt{bvar}\ 2\ 0})\ (\texttt{bvar}\ 0\ 1).$$

But then in u the bound variables referring to the outermost abstraction (shown squared) point to nowhere. Therefore, we consider $u^{[x]}$ instead of u, where

$$
\begin{array}{l}
u^{[x]} = \{0 \to x\}(\texttt{let bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in app}\ (\texttt{abs bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1)) \\
\quad = \texttt{let}\{1 \to x\}(\texttt{bvar}\ 0\,1, \texttt{bvar}\ 1\,0)\ \texttt{in}\{1 \to x\}(\texttt{app}\,(\texttt{abs bvar}\ 2\,0)(\texttt{bvar}\ 0\,1)) \\
\quad = \texttt{let bvar}\ 0\ 1, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs}\ \{2 \to x\}(\texttt{bvar}\ 2\ 0))\ (\texttt{bvar}\ 0\ 1) \\
\quad = \texttt{let bvar}\ 0\ 1, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar}\ 0\ 1)
\end{array}
$$

□

[1] Multiple binders are defined in [5]. Two constructions are given: One for non-recursive local declarations, and another for mutually recursive expressions. Yet both extensions are not completely developed.

[2] Elements in lists are numbered starting with 0 to match bound variables indices.

$$
\begin{array}{ll}
\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & = \texttt{bvar}\ i\ j \\
\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) & = \begin{cases} \texttt{bvar}\ k\ j & \text{if } \exists j : 0 \leq j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise} \end{cases} \\
\{k \leftarrow \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}\ t) \\
\{k \leftarrow \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \leftarrow \overline{x}\}\ t)\ (\{k \leftarrow \overline{x}\}\ v) \\
\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}\ t)
\end{array}
$$

where $\{k \leftarrow \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \leftarrow \overline{x}\}\,\cdot)\ \overline{t}$.

Fig. 4. Variable closing

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct). The definition of variable closing is based on a more general function $\{k \leftarrow \overline{x}\}t$ (Figure 4), where k indicates the level of nesting of binders. Whenever a free variable $\texttt{fvar}\ x$ is encountered, x is looked up in $\overline{x}$. If x occurs in position j, then the free variable is replaced by the bound variable $\texttt{bvar}\ k\ j$, otherwise it is left unchanged. Variable closing is then defined as ${}^{\backslash\overline{x}}t = \{0 \leftarrow \overline{x}\}t$. And its extension to lists is: ${}^{\backslash\overline{x}}\overline{t} = \texttt{List.map}\ ({}^{\backslash\overline{x}}\cdot)\ \overline{t}$.

Example 3. Now we close the term obtained by opening u in Example 2.
Let $t \equiv \texttt{let bvar}\ 0\ 1, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar}\ 0\ 1)$.

$$
\begin{array}{ll}
{}^{\backslash x}t = \{0 \leftarrow x\}(\texttt{let}\ \{\texttt{bvar}\ 0\ 1, \texttt{fvar}\ x\}\ \texttt{in app}\ (\texttt{abs}\ (\texttt{fvar}\ x))\ (\texttt{bvar}\ 0\ 1)) \\
\quad = \texttt{let}\ \{1 \leftarrow x\}(\texttt{bvar}\ 0\ 1, \texttt{fvar}\ x) \\
\qquad \texttt{in}\ \{1 \leftarrow x\}(\texttt{app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar}\ 0\ 1)) \\
\quad = \texttt{let bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in app}\ (\texttt{abs}\ \{2 \leftarrow x\}(\texttt{fvar}\ x))\ (\texttt{bvar}\ 0\ 1) \\
\quad = \texttt{let bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in app}\ (\texttt{abs bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1)
\end{array}
$$

Notice that the closed term coincides with u, the body of the abstraction in Example 2, although this is not always the case. □

2.3 Local Closure, Free Variables and Substitution

The locally nameless syntax in Figure 2 allows to build terms that have no corresponding expression in *Exp* (Figure 1). For instance, in $\texttt{abs}\ (\texttt{bvar}\ 1\ 5)$ index 1 does not refer to a binder in the term. Well-formed terms, i.e., those matching expressions in *Exp*, are called *locally closed*. To determine if a term is locally closed one should check that every bound variable has valid indices, i.e., that they refer to binders in the term. An easier method is to open with fresh names every abstraction and `let`-declaration in the term to be checked, and verify that no bound variable is reached. This checking is implemented with the *local closure* predicate `lc` given in Figure 5.

Observe that we use cofinite quantification (as introduced by Aydemir et al. in [1]) in the rules for the binders, i.e., abstraction and `let`. Cofinite quantification is an elegant alternative to exist-fresh conditions and provides stronger induction

$$\text{LC\_VAR}\ \frac{}{\texttt{lc}\ (\texttt{fvar}\ x)} \qquad \text{LC\_ABS}\ \frac{\forall x \notin L \subseteq Id \quad \texttt{lc}\ t^{[x]}}{\texttt{lc}\ (\texttt{abs}\ t)}$$

$$\text{LC\_APP}\ \frac{\texttt{lc}\ t \quad \texttt{lc}\ v}{\texttt{lc}\ (\texttt{app}\ t\ v)} \qquad \text{LC\_LET}\ \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id \quad \texttt{lc}\ [t : \overline{t}]^{\overline{x}}}{\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LC\_LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc}\ \cdot)\ \overline{t}}{\texttt{lc}\ \overline{t}}$$

Fig. 5. Local closure

$$\text{LCK-BVAR}\ \frac{i < k \wedge j < \texttt{List.nth}\ i\ \overline{n}}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)} \qquad \text{LCK-APP}\ \frac{\texttt{lc\_at}\ k\ \overline{n}\ t \qquad \texttt{lc\_at}\ k\ \overline{n}\ v}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t\ v)}$$

$$\text{LCK-FVAR}\ \frac{}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{fvar}\ x)} \qquad \text{LCK-LET}\ \frac{\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ [t : \overline{t}]}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LCK-ABS}\ \frac{\texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ t}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t)} \qquad \text{LCK-LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc\_at}\ k\ \overline{n}\ \cdot)\ \overline{t}}{\texttt{lc\_at}\ k\ \overline{n}\ \overline{t}}$$

Fig. 6. Local closure at level k

and inversion principles. Proofs are simplified, because it is not required to define exactly the set of fresh names (several examples of this are given in [5]). The rule LC-ABS establishes that an abstraction is locally closed if there exists a finite set of names L such that, for any name x not in L, the term $t^{[x]}$ is locally closed. Similarly, in the rule LC-LET we write $\overline{x}^{|\overline{t}|} \notin L$ to indicate that the list of distinct names $\overline{x}$ of length $|\overline{t}|$ are not in the finite set L. For any list $\overline{x}$ satisfying this condition, the opening of each term in the list of local declarations, $\overline{t}^{\overline{x}}$, and of the term affected by these declarations, $t^{\overline{x}}$, are locally closed. Notice that we have overloaded the predicate `lc` to work both on terms and list of terms. In the following we will overload other predicates and functions similarly. We write $[t : \overline{t}]$ for the list with head t and tail $\overline{t}$. In the following, $[\,]$ represents the empty list, $[t]$ is a unitary list, and $+\!\!+$ is the concatenation of lists.

We define a new predicate that checks if indices in bound variables are valid from a given level: *t is closed at level k*, written $\texttt{lc\_at}\ k\ \overline{n}\ t$ (Figure 6). As usual, k indicates the current depth, that is, how many binders have been passed by. Since binders can be either abstractions or local declarations, we need to keep the size of each binder (1 in the case of an abstraction, n for a `let` with n local declarations). These sizes are collected in the list $\overline{n}$, thus $|\overline{n}|$ should be at least k. A bound variable $\texttt{bvar}\ i\ j$ is closed at level k if i is smaller than k and j is smaller than $\texttt{List.nth}\ i\ \overline{n}$. The list $\overline{n}$ is new with respect to [5] because there the predicate `lc_at` is not defined for multiple binders.

It can be proved that if t is locally closed at level k for a given list of numbers $\overline{n}$, then it is also locally closed at level k for any list of numbers greater than $\overline{n}$.

Lemma 1. LC_AT_M_FROM_N $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \Rightarrow \forall \overline{m} \geq \overline{n}\,.\,\texttt{lc\_at}\ k\ \overline{m}\ t$

Where $\overline{m} \geq \overline{n}$ is the pointwise lifting to lists of the usual ordering on naturals.

$$
\begin{array}{lll}
\texttt{fv}(\texttt{bvar}\ i\ j) & = \emptyset & (\texttt{bvar}\ i\ j)[z/y] = \texttt{bvar}\ i\ j \\
\texttt{fv}(\texttt{fvar}\ x) & = \{x\} & (\texttt{fvar}\ x)[z/y] = \begin{cases} \texttt{fvar}\ z & \text{if } x = y \\ \texttt{fvar}\ x & \text{if } x \neq y \end{cases} \\
\texttt{fv}(\texttt{abs}\ t) & = \texttt{fv}(t) & (\texttt{abs}\ t)[z/y] = \texttt{abs}\ t[z/y] \\
\texttt{fv}(\texttt{app}\ t\ v) & = \texttt{fv}(t) \cup \texttt{fv}(v) & (\texttt{app}\ t\ v)[z/y] = \texttt{app}\ t[z/y]\ v[z/y] \\
\texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{fv}(\overline{t}) \cup \texttt{fv}(t) & (\texttt{let}\ \overline{t}\ \texttt{in}\ t)[z/y] = \texttt{let}\ \overline{t}[z/y]\ \texttt{in}\ t[z/y]
\end{array}
$$

where $\texttt{fv}(\overline{t}) = \texttt{List.foldright}\ (\cdot \cup \cdot)\ \emptyset\ (\texttt{List.map}\ \texttt{fv}\ \overline{t})$
$\overline{t}[z/y] = \texttt{List.map}\ ([z/y]\cdot)\ \overline{t}.$

Fig. 7. Free variables and substitution

The two approaches for local closure are equivalent, so that it can be proved that a term is locally closed if and only if it is closed at level 0.

Lemma 2. LC_IIF_LC_AT $\quad \texttt{lc}\ t \Leftrightarrow \texttt{lc\_at}\ 0\ [\,]\ t$

If the opening of a term is locally closed then the opening of the term with a different variable is locally closed too.

Lemma 3. LC_OP $\quad \texttt{lc}\ t^{[x]} \Rightarrow \texttt{lc}\ t^{[y]}$

Computing the *free variables* of a term t is very easy in the locally nameless representation, since bound and free variables are syntactically different. The set of free variables of $t \in \mathit{LNExp}$ is denoted as $\texttt{fv}(t)$, and it is defined in Figure 7.

A name x is said to be *fresh for a term* t, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t. Similarly for a list of distict names $\overline{x}$:

$$\frac{x \notin \texttt{fv}(t)}{\texttt{fresh}\ x\ \texttt{in}\ t} \qquad \frac{\overline{x} \notin \texttt{fv}(t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ t}$$

A term t is *closed* if it has no free variables at all:

$$\frac{\texttt{fv}(t) = \emptyset}{\texttt{closed}\ t}$$

Substitution replaces a variable name by another. For $t \in \mathit{LNExp}$ and $z, y \in \mathit{Id}$, $t[z/y]$ is the term where z substitutes any occurrence of y in t (see Figure 7).

Under some conditions variable closing and variable opening are inverse operations. More precisely, opening a term with fresh names and closing it with the same names, produces the original term. Symmetrically, closing a locally closed term and then opening it with the same names gives back the initial term.

Lemma 4.

CLOSE_OPEN_VAR $\quad \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow {}^{\backslash \overline{x}}(t^{\overline{x}}) = t$

OPEN_CLOSE_VAR $\quad \texttt{lc}\ t \Rightarrow ({}^{\backslash \overline{x}}t)^{\overline{x}} = t$

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : w}$$

Fig. 8. Natural semantics

3 Natural Semantics for Lazy λ-Calculus

The semantics defined by Launchbury in [7] follows a lazy strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : w$, that is, the expression $e \in \mathit{Exp}$ in the context of the heap Γ reduces to the value w in the context of the heap Δ. *Values* ($w \in \mathit{Val}$) are expressions in weak-head-normal-form (*whnf*). *Heaps* are partial functions from variables into expressions. Each pair (variable, expression) is called a *binding*, and it is represented by $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings may be updated to their corresponding computed values. The rules of this natural semantics are shown in Figure 8. The normalization of the λ-calculus, that has been mentioned in Section 2, simplifies the definition of the operational rules, although a renaming is still needed ($\hat{w}$ in Var) to avoid name clashing. This renaming is justified by Barendregt's variable convention [3].

Example 4. Without the renaming in rule Var heaps may end up binding a same name more than once. Take for instance the evaluation of the expression $e \equiv \texttt{let}\ x_1 = \lambda y.(\texttt{let}\ z = \lambda v.y\ \texttt{in}\ y), x_2 = (x_1\ x_3), x_3 = (x_1\ x_4), x_4 = \lambda s.s\ \texttt{in}\ x_2$ in the context of the empty heap. The evaluation of e implies the evaluation of x_2, and then the evaluation of $(x_1 x_3)$. This application leads to the addition of z to the heap bound to $\lambda v.x_3$. Subsequently, the evaluation of x_3 implies the evaluation of $(x_1 x_4)$. Without a renaming of values, variable z is added again to the heap, now bound to $\lambda v.x_4$. □

Theorem 1 in [7] states that "every heap/term pair occurring in the proof of a reduction is *distinctly named*", but we have found that the renaming fails to ensure this property. At least, it depends on how much fresh is this renaming.

Example 5. Let us evaluate in the context of the empty heap the expression

$$e \equiv \texttt{let}\ x_1 = (x_2\ x_3), x_2 = \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 = \lambda s.s\ \texttt{in}\ x_1$$

$$\{\,\} : e$$

$$\textsc{Let}\ \Big|\ \{x_1 \mapsto (x_2\ x_3), x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : x_1$$

$$\textsc{Var}\ \Big|\ \{x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : (x_2\ x_3)$$

$$\textsc{App}\ \Big|\ \{x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : x_2$$

$$\textsc{Var}\ \Big|\ \{x_3 \mapsto \lambda s.s\} : \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y$$

$$\textsc{Lam}\ \Big|$$

$$\vdots \qquad \{x_3 \mapsto \lambda s.s\} : \boxed{\lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y}$$

At this point the rule VAR requires to rename the value highlighted in the square. Notice that x_1 is fresh in the actual heap/term pair, and hence can be chosen to rename y. This would lead later in the derivation to introduce twice x_1 in the heap. The solution is to consider the condition of freshness in the whole derivation. This notion has not been formally defined by Launchbury. □

3.1 Locally Nameless Heaps

Before translating the semantic rules in Figure 8 to the locally nameless representation defined in Section 2, we have to establish how *bindings* and *heaps* are represented in this notation.

Recall that bindings associate expressions to free variables, therefore bindings are now pairs $(\texttt{fvar}\ x, t)$ with $x \in Id$ and $t \in LNExp$. To simplify, we will just write $x \mapsto t$. In the following, we will represent a heap $\{x_i \mapsto t_i\}_{i=1}^{n}$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names that are bound in the heap.

$$\texttt{dom}(\emptyset) = \emptyset \qquad\qquad \texttt{dom}(\Gamma, x \mapsto t) = \texttt{dom}(\Gamma) \cup \{x\}$$

In a well-formed heap names are defined at most once and terms are locally closed. The predicate `ok` expresses that a heap is well-formed:

$$\text{OK-EMPTY}\ \frac{}{\texttt{ok}\ \emptyset} \qquad\qquad \text{OK-CONS}\ \frac{\texttt{ok}\ \Gamma \qquad x \notin \texttt{dom}(\Gamma) \qquad \texttt{lc}\ t}{\texttt{ok}\ (\Gamma, x \mapsto t)}$$

The function `names` returns the set of names that appear in a heap, i.e., the names occurring in the domain or in the right-hand side terms:

$$\texttt{names}(\emptyset) = \emptyset \qquad\qquad \texttt{names}(\Gamma, x \mapsto t) = \texttt{names}(\Gamma) \cup \{x\} \cup \texttt{fv}(t)$$

This definition can be extended to (heap: term) pairs:

$$\texttt{names}(\Gamma : t) = \texttt{names}(\Gamma) \cup \texttt{fv}(t)$$

Next we define the freshness predicate of a list of names in a (heap:term) pair:

$$\frac{\overline{x} \notin \texttt{names}(\Gamma : t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : t)}$$

Substitution of variable names is extended to heaps as follows:

$$\emptyset[z/y] = \emptyset \qquad\qquad (\Gamma, x \mapsto t)[z/y] = (\Gamma[z/y], x[z/y] \mapsto t[z/y])$$
$$\textit{where}\ x[z/y] = \begin{cases} z & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

The following property is verified:

Lemma 5. OK_SUBS_OK $\qquad \texttt{ok}\ \Gamma \wedge y \notin \texttt{dom}(\Gamma) \Rightarrow \texttt{ok}\ \Gamma[y/x]$

$$\text{LNLam} \quad \frac{\{\texttt{ok}\ \Gamma\} \qquad \{\texttt{lc}\ (\texttt{abs}\ t)\}}{\Gamma : \texttt{abs}\ t \Downarrow \Gamma : \texttt{abs}\ t}$$

$$\text{LNVar} \quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \cup \texttt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : (\texttt{fvar}\ x) \Downarrow (\Delta, x \mapsto w) : w}$$

$$\text{LNApp} \quad \frac{\Gamma : t \Downarrow \Theta : \texttt{abs}\ u \qquad \Theta : u^{[x]} \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin \texttt{dom}(\Delta)\}}{\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x) \Downarrow \Delta : w}$$

$$\text{LNLet} \quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \qquad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Fig. 9. Locally nameless natural semantics

3.2 Locally Nameless Semantics

Once the locally nameless syntax and the corresponding operations, functions and predicates have been defined, three steps are sufficient to translate an inductive definition on λ-terms from the named representation into the locally nameless notation (as it is explained in [5]):

1. Replace the named binders, i.e., abstractions and let-constructions, with nameless binders by opening the bodies.
2. Cofinitely quantify the names introduced for variable opening.
3. Add premises to inductive rules in order to ensure that inductive judgements are restricted to locally closed terms.

We apply these steps to the inductive rules for the lazy natural semantics given in Figure 8. These rules produce judgements involving λ-terms as well as heaps. Hence, we also add premises that ensure that inductive judgements are restricted to well-formed heaps. The rules using the locally nameless representation are shown in Figure 9. For clarity, in the rules we put in braces the side-conditions to distinguish them better from the judgements.

The main difference with the rules in Figure 8 is the rule LNLet. To evaluate $\texttt{let}\ \overline{t}\ \texttt{in}\ t$ the local terms in $\overline{t}$ have to be introduced in the heap, so that the body t is evaluated in this new context. To this purpose fresh names $\overline{x}$ are needed to open the local terms and the body. The evaluation of $t^{\overline{x}}$ produces a final heap and a value. Both are dependent on the names chosen for the local variables. The domain of the final heap consists of the local names $\overline{x}$ and the rest of names, say $\overline{z}$. The rule LNLet is cofinite quantified. As it is explained in [5], the advantage of the cofinite rules over existential and universal ones is that the freshness side-conditions are not explicit. In our case, the freshness condition for $\overline{x}$ is *hidden* in the finite set L, which includes the names that should be avoided during the reduction. The novelty of our cofinite rule, compared with the ones appearing in [1] and [5] (that are similar to the cofinite rules for the predicate `lc` in Figure 5), is that the names introduced in the (infinite) premises do appear in the conclusion too. Therefore, in the conclusion of the rule LNLet we can replace the names $\overline{x}$ by any list $\overline{y}$ not in L.

The problem with explicit freshness conditions is that they are associated just to rule instances, while they should apply to the whole reduction proof. Take for instance the rule LNVar. In the premise the binding $x \mapsto t$ does no longer belong to the heap. Hence, a valid reduction for this premise may chose x as fresh (this corresponds to the problem shown in Example 5). We avoid this situation by requiring that x remains undefined in the final heap too. By contrast to the rule Var in Figure 8, no renaming of the final value w is needed.

The side-condition of rule LNApp deserves an explanation too. Let us suppose that x is undefined in the initial heap Γ. We must avoid that x is chosen as a fresh name during the evaluation of t. For this reason we require that x is defined in the final heap Δ only if x was already defined in Γ. Notice how the body of the abstraction, that is u, is open with the name x. This is equivalent to the substitution of x for y in the body of the abstraction $\lambda y.e'$ (see rule App in Figure 8).

A *regularity* lemma ensures that the judgements produced by this reduction system involve only well-formed heaps and locally closed terms.

Lemma 6.
REGULARITY $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \wedge \mathtt{ok}\ \Delta \wedge \mathtt{lc}\ w$

Similarly, Theorem 1 in [7] ensures that the property of being *distinctly named* is preserved by the rules in Figure 8. However, as shown in Example 5, the correctness of this result requires that freshness is relative to whole reduction proofs instead to the scope of rules.

The next lemma states that names defined in a context heap remain defined after the evaluation of any term in that context.

Lemma 7.
DEF_NOT_LOST $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta)$

Furthermore, fresh names are introduced only by the rule LNLet and, by the previous lemma, they remain bound in the final (heap: value) pair. Hence, any free variable appearing in a final (heap: value) pair is undefined only if the variable already occurs in the initial (heap: term) pair.

Lemma 8.
ADD_VARS $\quad \Gamma : t \Downarrow \Delta : w$
$\Rightarrow (x \in \mathtt{names}(\Delta : w) \Rightarrow (x \in \mathtt{dom}(\Delta) \vee x \in \mathtt{names}(\Gamma : t)))$

A *renaming* lemma ensures that the evaluation of a term is independent of the fresh names chosen in the reduction process. Moreover, any name in the context can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reduction proofs for (heap: term) pairs are unique up to renaming of the variables defined in the context heap.

Lemma 9.
RENAMING $\quad \Gamma : t \Downarrow \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w)$
$\Rightarrow \Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$

In addition, the renaming lemma permits to prove an *introduction* lemma for the cofinite rule LNLet which establishes that the corresponding existential rule is admissible too.

Lemma 10.

LET_INTRO $(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t)$
$\Rightarrow \Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}}$

This result, together with the renaming lemma, justifies that our rule LNLet is equivalent to Launchbury's rule Let used with normalized terms.

4 Related Work

In order to avoid α-conversion, we first considered a nameless representation like the de Bruijn notation [6], where variable names are removed and replaced by natural numbers. But this notation has several drawbacks. First of all, the de Bruijn representation is hard to read for humans. Even if we intend to check our results with some proof assistant like Coq [4], human readability helps intuition. At a more technical level, the de Bruijn notation does not have a good way to handle free variables, which are represented by indices, alike to bound variables. This is a serious weakness for our application. Recall that Launchbury's semantics uses contexts heaps that collect the bindings for the free variables that may occur in the term under evaluation. Any change in the domain of a heap, i.e., adding or deleting a binding, would lead to a shifting of the indices, thus complicating the statement and proof of results. Therefore, we prefer the more manageable locally nameless representation, where bound variable names are replaced by indices but free variables keep their names. This mixed notation combines the advantages of both named and nameless representations. On the one hand, α-conversion is avoided all the same. On the other hand, terms stay readable and easy to manipulate.

There exists in the literature different proposals for a locally nameless representation, and many works using these representations. Charguéraud offers in [5] a brief survey on these works, that we recommend to the interested reader.

Launchbury (implicitly) assumes Barendregt's variable convention [3] twice in [7]. First when he defines his operational semantics only for normalized λ-terms (i.e. every binder in a term binds a distinct name, which is also distinct from any free variable); and second, when he requires a (fresh) renaming of the values in the rule Var (see Figure 8). Urban, Berghofer and Norrish propose in [15] a method to strengthen an induction principle (corresponding to some inductive relation), so that Barendregt's variable convention comes already built in the principle. Unfortunately, we cannot apply these ideas to Launchbury's semantics, because the semantic rules (shown in Figure 8) do not satisfy the conditions that guarantee the *variable convention compatibility*, as described in [15]. In fact, as we have already pointed out, Launchbury's Theorem 1 (in [7]) is only correct if the renaming required in each application of the rule Var is fresh in the whole reduction proof. Therefore, we cannot use directly Urban's nominal package for

Isabelle/HOL [14] (including its recent extensions for general bindings described in [16]).

Nevertheless, Urban et al. achieve the "inclusion" of the variable convention in an induction principle by adding to each induction rule a side condition which expresses that the set of *bound* variables (i.e., those that appear in a binding position in the rule) are fresh in some *induction context* ([15]). Furthermore, this context is required to be finitely supported. This is closely related to the cofinite quantification that we have used for the rule LNLet in Figure 9. Besides, one important condition to ensure the variable convention compatibility is the *equivariance* of the functions and predicates occurring in the induction rules. Equivariance is a notion from nominal logic [10]. A relation is equivariant if it is preserved by permutation of names. Although we have not proven that the reduction relation defined by the rules in Figure 9 is equivariant, our *renaming lemma* (Lemma 9) establishes a similar result, that is, the reduction relation is preserved by (fresh) renaming.

5 Conclusions

We have used a more modern approach to binding, i.e., a locally nameless representation for the λ-calculus extended with mutually recursive local declarations. With this representation the reduction rule for local declarations implies the introduction of fresh names. We have used neither an existential nor a universal rule for this case. Instead, we have opted for a cofinite rule as introduced by Aydemir et al. in [1]. Freshness conditions are usually considered in each rule individually. Nevertheless, this technique produces name clashing when considering whole reduction proofs. A solution might be to decorate judgements with the set of forbidden names and indicate how to modify this set during the reduction process (this approach has been taken by Sestoft in [13]). However, this could be too restrictive in many occasions. Besides, existential rules are not easy to deal with because each reduction is obtained just for one specific list of names. If any of the names in this list causes a name clashing with other reduction proofs, then it is cumbersome to demonstrate that an alternative reduction for a fresh list does exist. Cofinite quantification has allowed us to solve this problem because in a single step reductions are guaranteed for an infinite number of lists of names. Nonetheless, our introduction lemma (Lemma 10) guarantees that a more conventional exists-fresh rule is correct in our reduction system too.

The cofinite quantification that we have used in our semantic rules is more complex than those in [1] and [5]. Our cofinite rule LNLet in Figure 9 introduces quantified variables in the conclusion as well, as the latter depends on the chosen names.

Compared to Launchbury's original semantic rules, our locally nameless rules include several extra side-conditions. Some of these conditions require that heaps and terms are well-formed (like in rule LNLam). The rest of side-conditions express restrictions on the choice of fresh names. These restrictions, together with the cofinite quantification, fix the problem with the renaming in rule Var that we have shown in Example 5.

For our locally nameless semantics we have shown a *regularity lemma* (Lemma 6) which ensures that every term and heap involved in a reduction proof is well-formed, and with a *renaming lemma* (Lemma 9) which indicates that the choice of names (free variables) is irrelevant as long as they are fresh enough. A heap may be seen as a multiple binder. Actually, the names defined (bound) in a heap can be replaced by other names, provided that terms keep their meaning in the context represented by the heap. Our renaming lemma ensures that whenever a heap is renamed with fresh names, reduction proofs are preserved. This renaming lemma is essential in rule induction proofs for some properties of the reduction system. More concretely, when one combines several reduction proofs coming from two or more premises in a reduction rule (for instance, in rule LNApp in Figure 9).

In summary, the contributions of this paper are:

1. A locally nameless representation of the λ-calculus extended with recursive local declarations;
2. A locally nameless version of the inductive rules of Launchbury's natural semantics for lazy evaluation;
3. A new version of cofinite rules where the variables quantified in the premises do appear in the conclusion too;
4. A set of interesting properties of our reduction system, including the regularity, the introduction and the renaming lemmas; and
5. A way to guarantee Barendregt's variable convention by redefining Launchbury's semantic rules with cofinite quantification and extra side-conditions.

6 Future Work

Our future tasks include the implementation in the proof assistant Coq [4] of the natural semantics redefined in this paper, and the formalization of the proofs for the lemmas given (regularity, renaming, introduction, etc.), which at present are just paper-and-pencil proofs. We will use this implementation to prove formally the equivalence of Launchbury's natural semantics with the alternative version given also in [7]. As we mentioned in Section 1, this alternative version differs from the original one in the introduction of indirections during β-reduction and the elimination of updates. At present we are working on the definition (using the locally nameless representation) of two intermediate semantics, one introducing indirections and the other without updates. Then, we will establish equivalence relations between the heaps obtained by each semantics, which makes able to prove the equivalence of the original natural semantics and the alternative one through the intermediate semantics.

Acknowledgments. This work is partially supported by the projects: TIN2009-14599-C03-01 and S2009/TIC-1465.

References

1. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: ACM Symposium on Principles of Programming Languages, POPL 2008, pp. 3–15. ACM Press (2008)
2. Baker-Finch, C., King, D., Trinder, P.W.: An operational semantics for parallel lazy evaluation. In: ACM-SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 162–173. ACM Press (2000)
3. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland (1984)
4. Bertot, Y.: Coq in a hurry. CoRR, abs/cs/0603118 (2006)
5. Charguéraud, A.: The locally nameless representation. Journal of Automated Reasoning, 1–46 (2011)
6. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 75(5), 381–392 (1972)
7. Launchbury, J.: A natural semantics for lazy evaluation. In: ACM Symposium on Principles of Programming Languages, POPL 1993, pp. 144–154. ACM Press (1993)
8. Nakata, K., Hasegawa, M.: Small-step and big-step semantics for call-by-need. CoRR, abs/0907.4640 (2009)
9. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
10. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation 186(2), 165–193 (2003)
11. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: An Operational Semantics for Distributed Lazy Evaluation. In: Trends in Functional Programming, vol. 10, pp. 65–80. Intellect (2010)
12. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: A locally nameless representation for a natural semantics for lazy evaluation. Technical Report 01/12, Dpt. Sistemas Informáticos y Computación. Universidad Complutense de Madrid (2012), `http://maude.sip.ucm.es/eden-semantics/`
13. Sestoft, P.: Deriving a lazy abstract machine. Journal of Functional Programming 7(3), 231–264 (1997)
14. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automatic Reasoning 40(4), 327–356 (2008)
15. Urban, C., Berghofer, S., Norrish, M.: Barendregt's Variable Convention in Rule Inductions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 35–50. Springer, Heidelberg (2007)
16. Urban, C., Kaliszyk, C.: General Bindings and Alpha-Equivalence in Nominal Isabelle. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 480–500. Springer, Heidelberg (2011)
17. van Eekelen, M., de Mol, M.: Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101. Radboud University Nijmegen (2007)

The Role of Indirections in Lazy Natural Semantics

Lidia Sánchez-Gil[1], Mercedes Hidalgo-Herrero[2], and Yolanda Ortega-Mallén[3](✉)

[1] Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
[2] Facultad de Educación, Universidad Complutense de Madrid, Madrid, Spain
[3] Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Madrid, Spain
yolanda@ucm.es

Abstract. Launchbury defines a natural semantics for lazy evaluation and proposes an alternative version which introduces indirections, eliminates blackholes and does not update closures. Equivalence of both semantics is not straightforward. In this paper we focus on the introduction of indirections during β-reduction and study how the heaps, i.e., the sets of bindings, obtained with this kind of evaluation do relate with the heaps produced by substitution. As a heap represents the context of evaluation for a term, we first define an equivalence that identifies terms with the same meaning under a given context. This notion of *context* equivalence is extended to heaps. Finally, we define a relation between heap/term pairs to establish the equivalence between Launchbury's alternative natural semantics and its corresponding version without indirections.

1 Motivation

More than twenty years have elapsed since Launchbury first presented in [9] a natural semantics for lazy evaluation (*call-by-need*), a key contribution to the semantic foundation for non-strict functional programming languages like Haskell or Clean. Throughout these years, Launchbury's semantics has been cited frequently and has inspired many further works as well as several extensions like in [2,8,10,13,17,20]. The success of Lanchbury's proposal resides in its simplicity. Expressions are evaluated with respect to a *context*, which is represented by a heap of *bindings*, that is, (variable, expression) pairs. This heap is explicitly managed to make possible the sharing of bindings, thus, modeling laziness.

In order to prove that this lazy (operational) semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury introduces some variations in the operational semantics. On the one hand, the update of bindings with their computed values is an operational notion without counterpart in the standard denotational semantics, so that the alternative natural semantics does no longer update bindings and becomes a *call-by-name* semantics. On the other hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds

A. Voronkov and I. Virbitskaite (Eds.): PSI 2014, LNCS 8974, pp. 288–303, 2015.
DOI: 10.1007/978-3-662-46823-4_24

to the actual argument. For a closer approach to this mechanism, in the alternative semantics applications are carried out by introducing *indirections*, i.e., variables bound to variables, instead of by performing the β-reduction through substitution. Besides, the denotation "undefined" indicates that there is no value associated to the expression being evaluated, but there is no indication of the reason for that. By contrast, in the operational semantics there are two possibilities for not reaching a value: either the reduction gets blocked if no rule is applicable, or the reduction never stops. The first case occurs in the original semantics when reducing self-references (*blackhole*). The rules in the alternative semantics guarantee that reductions never reach a blackhole.

Alas, the proof of the equivalence of the natural semantics and its alternative version is detailed nowhere, and a simple induction turns out to be insufficient. The *context-heap* semantics is too sensitive to the changes introduced by the alternative rules. Intuitively, both reduction systems should lead to the same results. However, this cannot be directly established since final values may contain free variables that are dependent on the context of evaluation, which is represented by the heap of bindings. The lack of update leads to the duplication of bindings, but is awkward to prove that duplicated bindings, as well as indirections, do not add relevant information to the context. Therefore, our challenge is to establish a way of relating the heaps and values obtained with each reduction system, and to prove that the semantics are equivalent, so that any reduction of a term in one of the systems has its counterpart in the other. To facilitate this task we consider separately the no updating and the introduction of indirections.

In this paper we investigate the effect of introducing indirections in a setting without updates, and we analyze the similarities and differences between the reductions proofs obtained with and without indirections. Indirections have also been used in [8] to model communication channels between processes.

We want to identify terms up to α-conversion, but dealing with α-equated terms usually implies the use of Barendregt's variable convention [3] to avoid the renaming of bound variables. However, the use of the variable convention is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [18]). Moreover, we intend to formalize our results with the help of some proof assistant like Coq [4] or Isabelle [11]. Looking for a binding system susceptible of formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [6]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [7], while free variables preserve their names. This is suitable in our case because context heaps collect free variables whose names we are interested in preserving in order to identify them more easily. A locally nameless version of Launchbury's natural semantics has been presented by the authors in [14,15].

Others are revisiting Launchbury's semantics too. For instance, Breitner has formally proven in [5] the correctness of the natural semantics by using Isabelle's nominal package [19], and presently he is working on the formalization of the adequacy. While Breitner is exclusively interested in formalizing the proofs, we have a broader objective: To analyze the effect of introducing indirections in the context heaps, and the correspondence between heap/value pairs obtained with

$$
\begin{array}{l}
x \in \mathit{Var} \\
e \in \mathit{Exp} ::= x \mid \lambda x.e \mid (e\ x) \mid \\
\qquad\qquad \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e
\end{array}
$$

(a) Named representation

$$
\begin{array}{lll}
x \in \mathit{Id} & & i, j \in \mathbb{N} \\
v \in \mathit{Var} & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in \mathit{LNExp} & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \\
 & & \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

(b) Locally nameless representation

Fig. 1. Extended λ-calculus

update and those produced without update. Furthermore, we want to prove the equivalence of the two operational semantics.

The main contributions of the present work are:

1. An equivalence relation to identify heaps that define the same free variables but whose corresponding closures may differ on *undefined free variables*;
2. A preorder that relates two heaps whenever the first can be transformed into the second by *eliminating indirections*;
3. An extension of the previous preorder relation for heap/term pairs expressing that two terms are equivalent if they have the same structure and their free variables, defined in the context of the respective heaps, are the same except for some indirections.
4. An equivalence theorem for Launchbury's alternative semantics and a version without indirections (and without update and blackholes).

The paper is structured as follows: In the next section we give a locally nameless version of Launchbury's semantics and its alternative rules. In Sect. 3 we define equivalence and preorder relations on terms, heaps and also on heap/term pairs. We include a number of interesting results concerning these relations and, finally, we prove the equivalence of Launchbury's alternative semantics and an intermediate semantics without update, without blackholes and without indirections. In the last section we draw conclusions and outline our future work.

2 A Locally Nameless Representation

The language described in [9] is a normalized lambda calculus extended with recursive local declarations. The abstract syntax, in the *named representation*, appears in Fig. 1a. Normalization is achieved in two steps. First an α-conversion is performed so that all bound variables have distinct names. In a second phase, it is ensured that arguments for applications are restricted to be variables. These static transformations make more explicit the sharing of closures and, thus, simplify the definition of the reduction rules.

Since there are two name binders, i.e., λ-abstraction and `let`-declaration, a quotient structure respect to α-equivalence is required. We avoid this by employing a *locally nameless representation* [6]. As mentioned above, our locally nameless representation has already been presented in [14,15]. Here we give only a brief overview avoiding those technicalities that are not essential to the contributions of the present work.

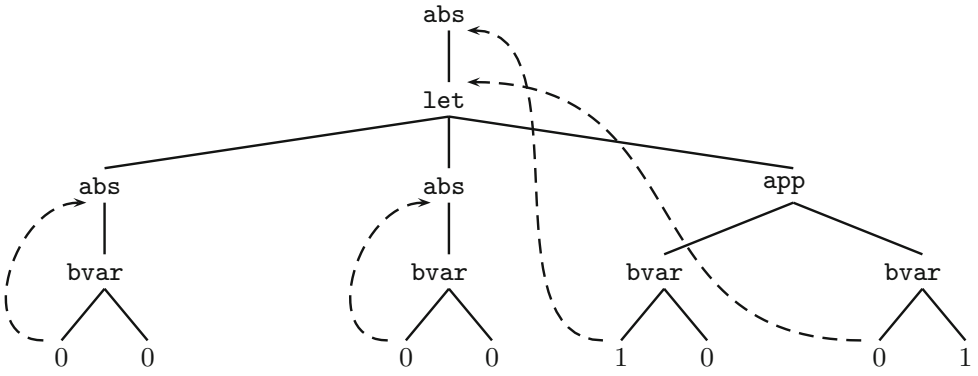


Fig. 2. Syntactic tree for a locally nameless term

2.1 Locally Nameless Syntax

The locally nameless version of the abstract syntax is shown in Fig. 1b. *Bound variables* and *free variables* are distinguished. Since `let`-declarations are multi-binders, we have followed Charguéraud [6] and bound variables are represented with two natural numbers: the first number is a de Bruijn index that counts how many binders (abstraction or `let`) have been passed through in the syntactic tree to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind one variable, so that the second number should be zero.

Example 1. Let $e \in Exp$ be a λ-expression given in the named representation: $e \equiv \lambda z.\texttt{let}\ \{x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2\}\ \texttt{in}\ (z\ x_2)$. The corresponding locally nameless term $t \in LNExp$ is:
$t \equiv \texttt{abs}\ (\texttt{let}\ \{\texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\}\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1))$.
Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms. The syntactic tree corresponding to t is shown in Fig. 2. □

This locally nameless syntax allows to build terms that have no corresponding named expression in *Exp*. For instance, when bound variables indices are out of range. Those terms in *LNExp* that do match expressions in *Exp* are called *locally-closed*, written $\texttt{lc}\ t$.

In the following, a list like $\{t_i\}_{i=1}^{n}$ is represented as $\overline{t}$, with length $|\overline{t}| = n$. Later on, we use $[t : \overline{t}]$ to represent a list with head t and tail $\overline{t}$; the empty list is represented as $[\,]$, a unitary list as $[t]$, and $+\!\!+$ stands for list concatenation.

We denote by $\texttt{fv}(t)$ the set of *free variables* of a term t. A name $x \in Id$ is *fresh in a term* $t \in LNExp$, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t, i.e., $x \notin \texttt{fv}(t)$. Similarly, for a list of names, $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t$ if $\overline{x} \notin \texttt{fv}(t)$, where $\overline{x}$ represents a list of pairwise-distinct names in Id. We say that two terms have the *same structure*, written $t \sim_S t'$, if they differ only in the names of their free variables.

Since there is no danger of name capture, *substitution* of variable names is trivial in the locally nameless representation. We write $t[y/x]$ for replacing the occurrences of x by y in the term t.

A *variable opening* operation is needed to manipulate locally nameless terms. This operation turns the outermost bound variables into free variables. The opening of a term $t \in \mathit{LNExp}$ with a list of names $\overline{x} \subseteq \mathit{Id}$ is denoted by $t^{\overline{x}}$. For simplicity, we write t^x for the variable opening with a unitary list $[x]$. We illustrate this concept and its use with an example:

Example 2. Let $t \equiv \texttt{abs}\ (\texttt{let bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in app}\ (\texttt{abs bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1))$. Hence, the body of the abstraction is:

$$u \equiv \texttt{let bvar}\ 0\ 1, \boxed{\texttt{bvar}\ 1\ 0}\ \texttt{in app}\ (\texttt{abs}\ \boxed{\texttt{bvar}\ 2\ 0}\ )\ (\texttt{bvar}\ 0\ 1).$$

But then in u the bound variables referring to the outermost abstraction of t (shown squared) point to nowhere. The opening of u with variable x replaces with x the bound variables referring to an hypothetical binder with body u: $u^x = \texttt{let bvar}\ 0\ 1, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar}\ 0\ 1)$. □

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct).

Example 3. We close the term obtained by opening u in Example 2.
Let $t \equiv \texttt{let bvar}\ 0\ 1, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar}\ 0\ 1)$, then
${}^{\backslash x}t = \texttt{let bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in app}\ (\texttt{abs bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1)$. □

Notice that in the last example the closed term coincides with u, the body of the abstraction in Example 2 that was opened with x, although this is not always the case. Only under some conditions variable closing and variable opening are inverse operations: If the variables are fresh in t, then ${}^{\backslash\overline{x}}(t^{\overline{x}}) = t$; and if the term is locally closed, then $({}^{\backslash\overline{x}}t)^{\overline{x}} = t$.

2.2 Locally Nameless Semantics

In the natural semantics defined by Launchbury [9] judgements are of the form $\Gamma : t \Downarrow \Delta : w$, that is, the term t in the context of the heap Γ reduces to the value w in the context of the (modified) heap Δ. *Values* ($w \in \mathit{Val}$) are terms in weak-head-normal-form (*whnf*) and *heaps* are collections of *bindings*, i.e., pairs (variable, term). A binding $(\texttt{fvar}\ x, t)$ with $x \in \mathit{Id}$ and $t \in \mathit{LNExp}$ is represented by $x \mapsto t$. In the following, we represent a heap $\{x_i \mapsto t_i\}_{i=1}^n$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names defined in the heap, so that $\texttt{dom}(\overline{x} \mapsto \overline{t}) = \overline{x}$. By contrast, the function `names` returns the set of all names that appear in a heap, i.e., the names occurring either in the domain or in the terms on the right-hand side of the bindings. This is used to define a freshness predicate for heaps: $\texttt{fresh}\ \overline{x}\ \texttt{in}\ \Gamma \stackrel{\texttt{def}}{=} \overline{x} \notin \texttt{names}(\Gamma)$.

$$\textsc{LNLam}\quad \frac{\{\mathtt{ok}\ \Gamma\} \qquad \{\mathtt{lc}\ (\mathtt{abs}\ t)\}}{\Gamma : \mathtt{abs}\ t \Downarrow \Gamma : \mathtt{abs}\ t}$$

$$\textsc{LNVar}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : \mathtt{fvar}\ x \Downarrow (\Delta, x \mapsto w) : w}$$

$$\textsc{LNApp}\quad \frac{\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \qquad \Theta : u^x \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin \mathtt{dom}(\Delta)\}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow \Delta : w}$$

$$\textsc{LNLet}\quad \frac{\begin{array}{c}\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id.[(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}}) : w^{\overline{x}} \\ \wedge\ {}^{\backslash \overline{x}}(\overline{s}^{\overline{x}}) = \overline{s} \wedge {}^{\backslash \overline{x}}(w^{\overline{x}}) = w] \\ \{\overline{y}^{|\overline{t}|} \notin L\}\end{array}}{\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{y}}) : w^{\overline{y}}}$$

Fig. 3. Natural semantics with locally nameless representation

$$\textsc{ALNVar}\quad \frac{(\Gamma, x \mapsto t) : t \Downarrow \Delta : w}{(\Gamma, x \mapsto t) : \mathtt{fvar}\ x \Downarrow \Delta : w}$$

$$\textsc{ALNApp}\quad \frac{\begin{array}{c}\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \\ \forall y \notin L \subseteq Id.[(\Theta, y \mapsto \mathtt{fvar}\ x) : u^y \Downarrow ([y : \overline{z}] \mapsto \overline{s}^y) : w^y \\ \wedge\ {}^{\backslash y}(\overline{s}^y) = \overline{s} \wedge {}^{\backslash y}(w^y) = w] \\ \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin [z : \overline{z}]\} \qquad \{z \notin L\}\end{array}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow ([z : \overline{z}] \mapsto \overline{s}^z) : w^z}$$

Fig. 4. Alternative rules with locally nameless representation

In a well-formed heap names are defined at most once and terms are locally closed. We write $\mathtt{ok}\ \Gamma$ to indicate that a heap is well-formed.

Figure 3 shows our locally nameless representation of the rules for the natural semantics for lazy evaluation. For clarity, in the rules we put in braces the side-conditions to better distinguish them from the judgements.

To prove the computational adequacy of the natural semantics with respect to a standard denotational semantics, Launchbury introduces alternative rules for variables and applications, whose locally nameless version is shown in Fig. 4. Observe that the ALNVar rule does not longer update the binding for the variable being evaluated, namely x. Besides, the binding for x does not disappear from the heap where the term bound to x is to be evaluated; therefore, any further reference to x leads to an infinite reduction. The effect of ALNApp is the addition of an indirection $y \mapsto \mathtt{fvar}\ x$ instead of performing the β-reduction by substitution, as in u^x in LNApp.

In the rules LNLet and ALNApp we use *cofinite quantification*, which is an alternative to "exists-fresh" quantification that provides stronger induction and inversion principles [1]. In LNLet the notation $\overline{x}^{|\overline{t}|} \notin L$ indicates that $\overline{x}$ is a list of length $|\overline{t}|$ of (distinct) names not belonging to the finite set L. Hence,

although there are not explicit freshness side-conditions in the rules, the finite set L represents somehow the names that should be avoided during a reduction proof. Among infinite possible combinations for $\overline{x}$, the set of names $\overline{y}$ is chosen for the reduction. The list $\overline{z}$ represents the rest of names defined in the heap which is obtained after the reduction. Notice how variable opening is used to express that the final heap and value may depend on the names that have been chosen. For instance, in LNLet, $w^{\overline{x}}$ indicates that it depends on the names $\overline{x}$, but there is a common basis w . Moreover, it is required that this basis does not contain occurrences of $\overline{x}$; this is expressed by ${}^{\backslash\overline{x}}(w^{\overline{x}}) = w$. A detailed explanation of these semantic rules can be found in [14–16].

In the following, the natural semantics (rules in Fig. 3) is referred as NS, and the alternative semantics (rules LNLam, LNLet and those in Fig. 4) as ANS. We write $\Downarrow^A$ for reductions in ANS. Launchbury proves in [9] the correctness of NS with respect to a standard denotational semantics, and a similar result for ANS is easily obtained (as the authors of this paper have done in [12]). Therefore, NS and ANS are "denotationally" equivalent in the sense that if an expression is reducible (in some heap context) by both semantics then the obtained values have the same denotation. But this is insufficient for our purposes, because we want to ensure that if for some (heap : term) pair a reduction exists in any of the semantics, then there must exist a reduction in the other too and the final heaps must be related. The changes in ANS might seem to involve no serious difficulties to prove the latter result. Unfortunately things are not so easy. On the one hand, the alternative rule for variables transforms the original call-by-need semantics into a call-by-name semantics because bindings are not updated and computed values are no longer shared. Moreover, in the original semantics the reduction of a self-reference gets blocked (*blackhole*), while in the alternative semantics self-references yield infinite reductions. On the other hand, the addition of indirections complicates the task of comparing the (heap : value) pairs obtained by each reduction system, as one may need to follow a chain of indirections to get the term bound to a variable. We deal separately with each modification and introduce two intermediate semantics: (1) the *No-update Natural Semantics* (NNS) with the rules of NS (Fig. 3) except for the variable rule, that corresponds to the one in the alternative version, i.e., ALNVar in Fig. 4; and (2) the *Indirection Natural Semantics* (INS) with the rules of NS but for the application rule, that corresponds to the alternative ALNApp rule in Fig. 4. We use $\Downarrow^N$ to represent reductions of NNS and $\Downarrow^I$ for those of INS.

The following table summarizes the characteristics of the four natural semantics defined above:

	NS - $\Downarrow$	**INS - $\Downarrow^I$**	**NNS - $\Downarrow^N$**	**ANS - $\Downarrow^A$**
Indirections	✗	✓	✗	✓
Update	✓	✓	✗	✗
Blackholes	✓	✓	✗	✗

It is guaranteed that the judgements produced by the locally nameless rules given in Figs. 3, 4 involve only well-formed heaps and locally closed terms. Furthermore, the reduction systems corresponding to these rules verify a number of interesting properties proved in [15]. We include here some new results that comprehend the alternative rules. In the four reduction systems, definitions are not lost during reduction, i.e., heaps only can grow with new names. But in the case of non updating (NNS and ANS) the bindings in the initial heap are preserved during the whole reduction:

Lemma 1. $\Gamma : t \Downarrow^K \Delta : w \Rightarrow \Gamma \subseteq \Delta$, *where* $\Downarrow^K$ *represents* $\Downarrow^N$ *and* $\Downarrow^A$.

During reduction, names might be added to the heap by the rules LNLet and ALNApp. However, there is no "spontaneous generation" of names, i.e., any name occurring in a final (heap : value) pair must either appear already in the initial (heap : term) pair or be defined in the final heap. The freshness of the names introduced by the rules LNLet and ALNApp is determined as follows:

Lemma 2. *1.* $\Gamma : t \Downarrow^N \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ \Gamma$.
2. $\Gamma : t \Downarrow^A \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ (\Gamma : t)$.

The following *renaming* lemma ensures that the evaluation of a term is independent of the names chosen during the reduction process. Further, any name defined in the context heap can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reductions for (heap : term) pairs are unique up to α-conversion of the names defined in the heap.

Lemma 3. *(Renaming)*

1. $\Gamma : t \Downarrow^K \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ \Gamma, \Delta, t, w \Rightarrow \Gamma[y/x] : t[y/x] \Downarrow^K \Delta[y/x] : w[y/x]$.
2. $\Gamma : t \Downarrow^K \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ \Gamma, \Delta, t, w \wedge x \notin \mathtt{dom}(\Gamma) \wedge x \in \mathtt{dom}(\Delta)$
   $\Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$,

where $\Gamma[y/x]$ indicates that name substitution is done in the left and right hand sides of the heap Γ, and $\Downarrow^K$ represents $\Downarrow$, $\Downarrow^A$, $\Downarrow^I$, and $\Downarrow^N$.

Detailed proofs are given in [15], and also in [16] that is an extended version of the present paper including detailed proofs for all the lemmas and propositions.

3 Indirections

The aim in this section is to prove the equivalence of NNS and ANS. After the evaluation of a term in a given context, each semantics yields a different binding heap. It is necessary to analyze their differences, which lie in the indirections introduced by ANS. An *indirection* is a binding of the form $x \mapsto \mathtt{fvar}\ y$, that is, it just redirects to another variable name. The set of indirections of a heap Γ is denoted by $\mathtt{Ind}(\Gamma)$.

Example 4. Let us evaluate $t \equiv \texttt{let abs (bvar 0 0) in app (abs } s\texttt{) (bvar 0 0)}$, where $s \equiv \texttt{let abs (bvar 0 0), app (bvar 0 0) (bvar 1 0) in abs (bvar 0 0)}$, in the empty context $\Gamma = \emptyset$:

$$\begin{aligned}
\Gamma : t \Downarrow^N\ & \{x_0 \mapsto \texttt{abs (bvar 0 0)}, x_1 \mapsto \texttt{abs (bvar 0 0)}, x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } x_0\texttt{)}\} \\
& : \texttt{abs (bvar 0 0)} \\
\Gamma : t \Downarrow^A\ & \{x_0 \mapsto \texttt{abs (bvar 0 0)}, x_1 \mapsto \texttt{abs (bvar 0 0)}, x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } y\texttt{)}, \\
& y \mapsto \texttt{(fvar } x_0\texttt{)}\} : \texttt{abs (bvar 0 0)}
\end{aligned}$$

The value produced is the same in both cases. Yet, when comparing the final heap in $\Downarrow^A$ with that in $\Downarrow^N$, we observe that there is an extra indirection, $y \mapsto \texttt{fvar } x_0$. This indirection corresponds to the binding introduced by ALNApp to reduce the application in the term t. □

The previous example gives a hint of how to establish a relation between the heaps that are obtained with NNS and those produced by ANS. Two heaps are related if one can be obtained from the other by eliminating some indirections. For this purpose we define how to remove indirections from a heap, while preserving the evaluation context represented by that heap.

$$\begin{aligned}
(\emptyset, x \mapsto \texttt{fvar } y) \ominus x &= \emptyset \\
((\Gamma, z \mapsto t), x \mapsto \texttt{fvar } y) \ominus x &= ((\Gamma, x \mapsto \texttt{fvar } y) \ominus x, z \mapsto t[y/x])
\end{aligned}$$

This is generalized to remove a sequence of indirections from a heap:

$$\Gamma \ominus [\,] = \Gamma \qquad\qquad \Gamma \ominus [x : \overline{x}] = (\Gamma \ominus x) \ominus \overline{x}$$

3.1 Context Equivalence

The meaning of a term depends on the meaning of its free variables. However, if a free variable is not defined in the context of evaluation of a term, then the name of this free variable is irrelevant. Therefore, we consider that two terms are equivalent in a given context if they only differ in the names of the free variables that do not belong to the context.

Definition 1. *Let $V \subseteq Id$, and $t, t' \in LNExp$. We say that t and t' are* context-equivalent *in V, written $t \approx^V t'$, when*

$$\text{CE-BVAR}\quad \frac{}{(\texttt{bvar } i\ j) \approx^V (\texttt{bvar } i\ j)} \qquad \text{CE-FVAR}\quad \frac{x, x' \notin V \vee x = x'}{(\texttt{fvar } x) \approx^V (\texttt{fvar } x')}$$

$$\text{CE-ABS}\quad \frac{t \approx^V t'}{(\texttt{abs } t) \approx^V (\texttt{abs } t')} \qquad \text{CE-APP}\quad \frac{t \approx^V t' \quad v \approx^V v'}{(\texttt{app } t\ v) \approx^V (\texttt{app } t'\ v')}$$

$$\text{CE-LET}\quad \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \approx^V \overline{t'} \quad t \approx^V t'}{(\texttt{let } \overline{t} \texttt{ in } t) \approx^V (\texttt{let } \overline{t'} \texttt{ in } t')}$$

Fixed the set of names V, $\approx^V$ is an equivalence relation on *LNExp*.

Proposition 1.

CE_REF	$t \approx^V t$
CE_SYM	$t \approx^V t' \Rightarrow t' \approx^V t$
CE_TRANS	$t \approx^V t' \wedge t' \approx^V t'' \Rightarrow t \approx^V t''$

Based on this equivalence on terms, we define a family of relations on heaps, where heaps are equivalent when they have the same domain and corresponding closures differ only in the free variables not defined in a given context:

Definition 2. *Let $V \subseteq Id$, and $\Gamma, \Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-context-equivalent *in V, written $\Gamma \approx^V \Gamma'$, when*

$$\text{HCE-EMPTY}\ \frac{}{\emptyset \approx^V \emptyset} \qquad \text{HCE-CONS}\ \frac{\Gamma \approx^V \Gamma' \quad t \approx^V t' \quad \mathtt{lc}\ t \quad x \notin \mathtt{dom}(\Gamma)}{(\Gamma, x \mapsto t) \approx^V (\Gamma', x \mapsto t')}$$

The relations defined above are equivalences on well-formed heaps.

Proposition 2.

HCE_REF	$\mathtt{ok}\ \Gamma \Rightarrow \Gamma \approx^V \Gamma$
HCE_SYM	$\Gamma \approx^V \Gamma' \Rightarrow \Gamma' \approx^V \Gamma$
HCE_TRANS	$\Gamma \approx^V \Gamma' \wedge \Gamma' \approx^V \Gamma'' \Rightarrow \Gamma \approx^V \Gamma''$

Moreover, if two heaps are heap-context-equivalent, then both are well-formed, have the same domain, and have the same indirections.

There is an alternative characterization for heap-context-equivalence which expresses that heaps are context-equivalent whenever they are well-formed, have the same domain, and each pair of corresponding bound terms is context-equivalent.

Lemma 4. $\Gamma \approx^V \Gamma' \Leftrightarrow$
$\mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma' \wedge \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \wedge (x \mapsto t \in \Gamma \wedge x \mapsto t' \in \Gamma' \Rightarrow t \approx^V t')$.

Considering context-equivalence on heaps, we are particularly interested in the case where the context coincides with the domain of the heaps:

Definition 3. *Let $\Gamma, \Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-equivalent, *written $\Gamma \approx \Gamma'$, if they are heap-context-equivalent in* $\mathtt{dom}(\Gamma)$, *i.e., $\Gamma \approx^{\mathtt{dom}(\Gamma)} \Gamma'$.*

The following lemmas establish the uniqueness (up to permutation) of the set of indirections that sets up the equivalence of two heaps. First, we have that the order in which two indirections are removed from a heap can be exchanged, producing equivalent heaps.

Lemma 5. $\mathtt{ok}\ \Gamma \wedge x, y \in \mathtt{Ind}(\Gamma) \wedge x \neq y \Rightarrow \Gamma \ominus [x, y] \approx \Gamma \ominus [y, x]$.

Next, the previous result is generalized so that any permutation of a sequence of indirections produces equivalent heaps. Moreover, if equivalent heaps are obtained by removing different sequences of indirections, then these must be the same up to permutation.

Lemma 6. $\mathtt{ok}\ \Gamma \wedge \overline{x}, \overline{y} \subseteq \mathtt{Ind}(\Gamma) \Rightarrow (\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \Leftrightarrow \overline{y} \in \mathcal{S}(\overline{x}))$,
where $\mathcal{S}(\overline{x})$ denotes the set of all permutations of $\overline{x}$.

3.2 Indirection Relation

Coming back to the idea of Example 4, where a heap can be obtained from another by removing some indirections, we define the following relation on heaps:

Definition 4. *Let $\Gamma, \Gamma' \in LNHeap$. We say that Γ is* indirection-related *to Γ', written $\Gamma \gtrsim_I \Gamma'$, when*

$$\text{IR-HE}\ \frac{\Gamma \approx \Gamma'}{\Gamma \gtrsim_I \Gamma'} \qquad \text{IR-IR}\ \frac{\mathtt{ok}\ \Gamma \quad \Gamma \ominus x \gtrsim_I \Gamma' \quad x \in \mathtt{Ind}(\Gamma)}{\Gamma \gtrsim_I \Gamma'}$$

There is an alternative characterization for the relation $\gtrsim_I$ which expresses that a heap is indirection-related to another whenever the later can be obtained from the former by removing a sequence of indirections.

Proposition 3. $\Gamma \gtrsim_I \Gamma' \Leftrightarrow \mathtt{ok}\ \Gamma \wedge \exists\, \overline{x} \subseteq \mathtt{Ind}(\Gamma)\,.\, \Gamma \ominus \overline{x} \approx \Gamma'$.

By Lemma 6, the sequence of indirections is unique up to permutations, and it corresponds to the difference between the domains of the related heaps.

Corollary 1. $\Gamma \gtrsim_I \Gamma' \Rightarrow \Gamma \ominus (\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')) \approx \Gamma'$.[1]

The *indirection-relation* is a preorder on the set of well-formed heaps.

Proposition 4.

$$\begin{array}{ll} \text{IR\_REF} & \mathtt{ok}\ \Gamma \Rightarrow \Gamma \gtrsim_I \Gamma \\ \text{IR\_TRANS} & \Gamma \gtrsim_I \Gamma' \wedge \Gamma' \gtrsim_I \Gamma'' \Rightarrow \Gamma \gtrsim_I \Gamma'' \end{array}$$

We extend Definition 4 to (heap : term) pairs. Again we use cofinite quantification instead of adding freshness conditions on the new name z.

Definition 5. *Let $\Gamma, \Gamma' \in LNHeap$, and $t, t' \in LNExp$. We say that $(\Gamma : t)$ is* indirection-related *to $(\Gamma' : t')$, written $(\Gamma : t) \gtrsim_I (\Gamma' : t')$, when*

$$\text{IR-HT}\ \frac{\forall z \notin L \subseteq Id.(\Gamma, z \mapsto t) \gtrsim_I (\Gamma', z \mapsto t')}{(\Gamma : t) \gtrsim_I (\Gamma' : t')}$$

We illustrate these definitions with an example.

Example 5. Let us consider the following heap and term:

$$\begin{array}{l} \Gamma = \{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0)), \\ \qquad y_0 \mapsto \mathtt{fvar}\ x_2\} \\ t\ = \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ \mathtt{bvar}\ 0\ 0) \end{array}$$

The (heap : term) pairs related with $(\Gamma : t)$ are obtained by removing the sequences of indirections $[\,]$, $[y_0]$, $[x_0]$, and $[x_0, y_0]$:

[1] Since the ordering of indirections is irrelevant, $\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$ represents any sequence with the names defined in Γ but undefined in Γ'.

(a) $\{x_0 \mapsto \texttt{fvar}\ x_1, x_1 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{bvar}\ 0\ 0)),$
$y_0 \mapsto \texttt{fvar}\ x_2\}$
$: \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{bvar}\ 0\ 0))$

(b) $\{x_0 \mapsto \texttt{fvar}\ x_1, x_1 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{bvar}\ 0\ 0))\}$
$: \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_0)\ (\texttt{bvar}\ 0\ 0))$

(c) $\{x_1 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_1)\ (\texttt{bvar}\ 0\ 0)), y_0 \mapsto \texttt{fvar}\ x_2\}$
$: \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_1)\ (\texttt{bvar}\ 0\ 0))$

(d) $\{x_1 \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), x_2 \mapsto \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_1)\ (\texttt{bvar}\ 0\ 0))\}$
$: \texttt{abs}\ (\texttt{app}\ (\texttt{fvar}\ x_1)\ (\texttt{bvar}\ 0\ 0))$ □

Notice that in Example 4 the (heap : term) pair obtained with ANS is indirection-related to the pair obtained with NNS by removing the indirection $y \mapsto \texttt{fvar}\ x$.

Now we are ready to establish the equivalence between ANS and NNS in the sense that if a reduction proof can be obtained with ANS for some term in a given context heap, then there must exist a reduction proof in NNS for the same (heap : term) pair such that the final (heap : value) is indirection-related to the final (heap : value) obtained with ANS, and vice versa.

Theorem 1 *(Equivalence ANS-NNS).*

1. $\Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in LNHeap\,.\,\exists w_N \in Val\,.\,\Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \succsim_I (\Delta_N : w_N)$.
2. $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in LNHeap\,.\,\exists w_A \in Val\,.\,\exists \overline{x} \subseteq \texttt{dom}(\Delta_N) - \texttt{dom}(\Gamma)\,.\,\exists \overline{y} \subseteq Id\,.$
$|\overline{x}| = |\overline{y}| \wedge \Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \succsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$.

Notice that in the second part of the theorem, i.e., from NNS to ANS, a renaming may be needed. This renaming only affects the names that are added to the heap during the reduction process. This is due to the fact that in NNS names occurring in the evaluation term (that is t in the theorem) may disappear during the evaluation and, consequently, they may be chosen on some application of the rule LNLet and added to the final heap. This cannot happen in ANS due to the introduction of indirections (see Lemma 2).

To prove this theorem by rule induction, a generalization is needed. Instead of evaluating the same term in the same initial heap, we consider indirection-related initial (heap : term) pairs:

Proposition 5. $(\Gamma_A : t_A) \succsim_I (\Gamma_N : t_N)$

1. $\forall \overline{x} \notin L \subseteq Id.[\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}_A{}^{\overline{x}}) = \overline{s}_A \ \wedge {}^{\backslash \overline{x}}(w_A{}^{\overline{x}}) = w_A]$
$\Rightarrow \exists \overline{y} \notin L\,.\,\exists \overline{s}_N \subset LNExp\,.\,\exists w_N \in LNVal\,.$
$\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}} \ \wedge {}^{\backslash \overline{z}}(\overline{s}_N{}^{\overline{z}}) = \overline{s}_N \ \wedge {}^{\backslash \overline{z}}(w_N{}^{\overline{z}}) = w_N \wedge \overline{z} \subseteq \overline{y} \wedge$
$((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$

2. $\forall \overline{x} \notin L \subseteq Id.[\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}_N{}^{\overline{x}}) = \overline{s}_N \wedge {}^{\backslash \overline{x}}(w_N{}^{\overline{x}}) = w_N]$
$\Rightarrow \exists \overline{z} \notin L\,.\,\exists \overline{s}_A \subset LNExp\,.\,\exists w_A \in LNVal\,.$
$\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}} \ \wedge {}^{\backslash \overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \ \wedge {}^{\backslash \overline{y}}(w_A{}^{\overline{y}}) = w_A \wedge \overline{z} \subseteq \overline{y} \wedge$
$((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$

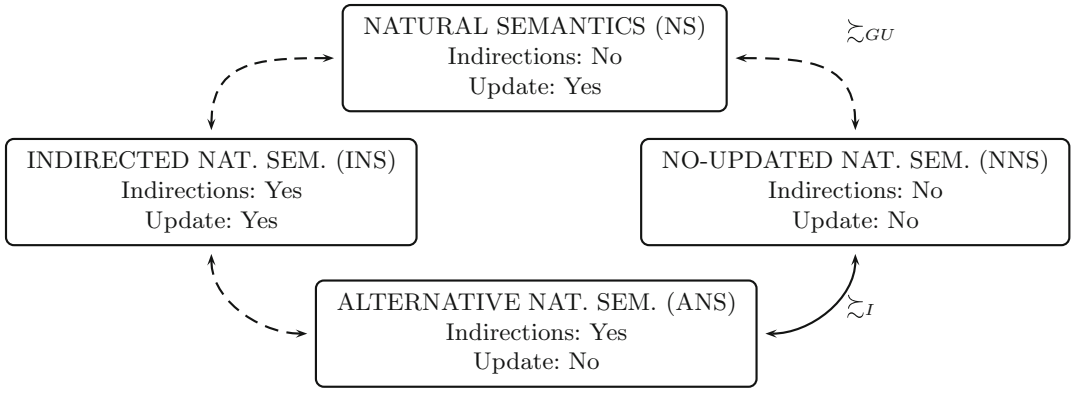


Fig. 5. The relations between the semantics

Once more, cofinite quantification replaces freshness conditions. For instance, in the second part of the proposition it is required that the names introduced during the reduction for NNS do not collide with names that are already defined in the initial heap for ANS. The cofinite quantification expresses that if there is an infinite number of "similar" reduction proofs for $(\Gamma_N : t_N)$, each introducing alternative names in the heap, one can chose a reduction proof such that the new bindings do not interfere with $(\Gamma_A : t_A)$.

Since there is update neither in ANS nor in NNS (Lemma 1), a final heap can be expressed as the initial heap plus some set of bindings, such as $(\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}})$. In this case, $\overline{x}$ represents the list of new names, i.e., those that have been added during the reduction of local declarations, as well as the indirections introduced by the alternative application rule. Since the terms bound to these new names are dependent on $\overline{x}$, they are represented as $\overline{s}_A{}^{\overline{x}}$. Similarly for the final value $w_A{}^{\overline{x}}$. The proposition indicates that it is possible to construct reductions for NNS whose set of new defined names is a subset of the set of new names of the corresponding ANS reduction (NNS only adds new names with the rule LNLet). Detailed proofs of the theorem and the proposition are given in [16].

4 Conclusions and Future Work

Launchbury natural semantics (NS) has turned out to be too much sensitive to the changes introduced by the alternative semantics (ANS), i.e., indirections and no-update. These changes should lead to the same values, but this cannot be directly established since values may contain free variables which are dependent on the context of evaluation, represented by the heap. And, precisely, the changes introduced by the ANS do affect deeply the heaps. In fact, the equivalence of the values produced by the NS and the ANS is based on their correctness with respect to a denotational semantics. Although indirections and duplicated bindings (consequence of not updating) do not add new information to the heap, it is not straightforward to prove it formally.

Since the variations introduced by Launchbury in the ANS do affect two rules, i.e. the variable rule (no-update) and the application rule (indirections),

we have defined two intermediate semantics to deal separately with the effect of each modification: The NNS (without update) and the INS (with indirections). A schema of the semantics and how to relate them is included in Fig. 5.

In this paper we have compared NNS with ANS, that is, substitution vs. indirections. We have started by defining an equivalence $\approx$ such that two heaps are considered equivalent when they have the same domain and the corresponding closures may differ only in the free variables not defined in the heaps. We have used this equivalence to define a preorder $\succsim_I$ expressing that a heap can be transformed into another by eliminating indirections. Furthermore, the relation has been extended to (heap : terms) pairs, expressing that two terms can be considered equivalent when they have the same structure and their free variables (only those defined in the context of the corresponding heap) are the same except for some indirections. We have used this extended relation to establish the equivalence between NNS and ANS (Theorem 1).

At present we are working on the equivalence of NS and NNS, which will close the path from NS to ANS. In order to compare NS with NNS, that is, update vs. no-update, new relations on heaps and terms have to be defined. No updating the bindings in the heap corresponds to a call-by-name strategy, and implies the duplication of evaluation work, that leads to the generation of duplicated bindings. More precisely, duplicated bindings come from several evaluations of the same `let`-declarations, so that they form *groups* of equivalent bindings. Therefore, we first define a preorder $\succsim_G$ that relates two heaps whenever the first can be transformed into the second by eliminating duplicated groups of bindings. Afterwards, we define a relation $\sim_U$ that establishes when a heap is an updated version of another heap. Finally, both relations must be combined to obtain the *group-update* relation $\succsim_{GU}$ that, extended for (heap : terms), will allow us to formulate an equivalence theorem for NS and NNS, similar to Theorem 1.

Although the relations $\succsim_I$ and $\succsim_{GU}$ are sufficient for proving the equivalence of NS and ANS, it would be interesting to complete the picture by comparing NS with INS, and then INS with ANS. For the first step, we have to define a preorder similar to $\succsim_I$, but taking into account that extra indirections may now be updated, thus leading to "redundant" bindings. For the second step, some version of the group-update relation is needed. Dashed lines indicate future work.

We have used a locally nameless representation to avoid problems with the α-equivalence, while keeping a readable formalization of the syntax and semantics. This representation allow us to deal with heaps in a convenient and easy way, avoiding the problems that arise when using the de Bruijn notation (indexes do change when bindings are introduced in or eliminated from heaps; moreover, the formalization becomes unreadable). We have also introduced cofinite quantification (in the style of [1]) in the evaluation rules that introduce fresh names, namely the rule for local declarations (LNLet) and for the alternative application (ALNApp). Moreover, this representation is more amenable to formalization in proof assistants. In fact we have started to implement the semantic rules given in Sect. 2.2 using Coq [4], with the intention of obtaining a formal checking of our proofs.

Acknowledgments. This work is partially supported by the projects: TIN2012-39391-C04-04 and S2009/TIC-1465.

References

1. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: ACM Symposium on Principles of Programming Languages, POPL 2008, pp. 3–15. ACM Press (2008)
2. Baker-Finch, C., King, D., Trinder, P.W.: An operational semantics for parallel lazy evaluation. In: ACM-SIGPLAN International Conference on Functional Programming (ICFP 2000), Montreal, Canada, pp. 162–173, September 2000
3. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam (1984)
4. Bertot, Y.: Coq in a hurry. CoRR, abs/cs/0603118 (2006)
5. Breitner, J.: The correctness of launchbury's natural semantics for lazy evaluation. Archive of Formal Proofs, January 2013. Formal proof development, Amended version May 2014. http://afp.sf.net/entries/Launchbury.shtml
6. Charguéraud, A.: The locally nameless representation. J. Autom. Reason. **46**(3), 363–408 (2012)
7. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indag. Math. **75**(5), 381–392 (1972)
8. Hidalgo-Herrero, M., Ortega-Mallén, Y.: An operational semantics for the parallel language Eden. Parallel Process. Lett. (World Scientific Publishing Company) **12**(2), 211–228 (2002)
9. Launchbury, J.: A natural semantics for lazy evaluation. In: ACM Symposium on Principles of Programming Languages, POPL 1993, pp. 144–154. ACM Press (1993)
10. Nakata, K., Hasegawa, M.: Small-step and big-step semantics for call-by-need. J. Funct. Program. **19**(6), 699–722 (2009)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: Call-by-need, call-by-name, and natural semantics. Technical report UU-CS-2010-020, Department of Information and Computing Sciences, Utrech University (2010)
13. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: An operational semantics for distributed lazy evaluation. In: Trends in Functional Programming, pp. 65–80, vol. 10. Intellect (2010)
14. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: A locally nameless representation for a natural semantics for lazy evaluation. Technical report 01/12, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid (2012). http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-1-12.pdf
15. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: A locally nameless representation for a natural semantics for lazy evaluation. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 105–119. Springer, Heidelberg (2012)

16. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: The role of indirections in lazy natural semantics (extended version). Technical report 13/13, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid (2013). http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/TR-13-13.pdf
17. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. **7**(3), 231–264 (1997)
18. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 35–50. Springer, Heidelberg (2007)
19. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in nominal Isabelle. Log. Methods Comput. Sci. **8**(2:14), 1–35 (2012)
20. van Eekelen, M., de Mol, M.: Proving lazy folklore with mixed lazy/strict semantics. In: Barendsen, E., Capretta, V., Geuvers, H., Niqui, M. (eds.) Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday, pp. 87–101. Radboud University Nijmegen (2007)

Chapter 8

An Operational Semantics for Distributed Lazy Evaluation

Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero, Yolanda Ortega-Mallén[1]
Category: Research

Abstract: We present a semantic model for distributed lazy evaluation where parallelism overrides laziness in terms of eager process creation and value communication. A small-step operational semantics defined with two transition levels: local rules for the internal behavior of each process, and global rules for process creation, interaction, and scheduling. This semantics is proved to be correct and computational adequate with respect to a standard denotational semantics. To obtain these results two intermediate semantics are defined: a 1-processor version of the small-step operational semantics and an extension of the natural semantics defined in [9].

8.1 INTRODUCTION

The increase of processors speed is reaching its limit, and other ways have to be explored to satisfy the voracity of computer applications. A promising alternative is computing in parallel. Parallel (and concurrent) programming is not certainly a novelty, but the low costs of computer components have contributed to the proliferation of parallel and distributed machines. Languages that ease the complex job of parallel programming are needed, and the functional paradigm is claimed to be a good candidate on account of its well-known advantages, such as abstraction, expressiveness, referential transparency, and a clean semantic model.

Referential transparency permits the implementation of alternative orders of execution while retaining the functionality of a program. This property, inherent to pure functional languages, is a key factor for the exploitation of parallelism in

[1]Universidad Complutense de Madrid, Spain; lidiasg@mat.ucm.es, mhidalgo@edu.ucm.es, yolanda@sip.ucm.es

the functional paradigm, ranging from a completely implicit parallelism or automatic parallelization, to an explicit parallelism where the programmer distributes the computation among a set of communicating processes that even may be located by the programmer himself at designated processors. In [10] an excellent classification of functional parallel approaches by level of control of parallelism can be found. Many of these approaches are extensions of sequential functional languages. For instance, Haskell has been used as the basis of a various set of parallel and distributed languages (see [15] for a comprehensive survey). One of these parallel extensions of Haskell is Eden [11] which includes a set of *coordination* features to control the parallel evaluation of processes while keeping the high-level nature of the declarative paradigm. The meaning of each construction of Eden is defined by using an abstract machine, DREAM [3]. However, this is a very low level semantics dealing with stacks and channels ports, and other semantics have been defined. Particularly, in [6, 5] is defined a small-step operational semantics with an intermediate level of abstraction, where the distribution of the computation among processes can be observed. In [6, 5], the main ideas for modelling a distributed computation —in Eden there is not shared memory— are described, but no semantic properties like *correctness* (reductions preserve the meaning of terms), *computational adequacy* (reduction corresponds to a non-bottom denotation) or *determinacy* (the final value obtained for a term is independent of the number of processors used during the evaluation) are given.

We therefore present a restricted and revised version of the semantics given in [6, 5] for a lazy and distributed λ-calculus, which includes a *parallel application* that instantiates a new process for computing the application speculatively. In our calculus there is not shared memory, hence the closures needed to compute the application have to be copied from the parent to the child process at the moment of creation. Under a proper notion of abstraction, where the underlying distributed process system is ignored and only the functional relation is observed, the operational semantics is proved to be correct and adequate with respect to a standard denotational semantics. We also prove determinacy for this operational semantics.

8.2 RELATED WORK

Author	**Language**	**Properties**
J. Launchbury (natural semantics)	λ-calculus + recursive let	sequential big step sharing
M.v. Eekelen & M. de Mol	strict let	sequential big step mix strict/lazy
Baker Finch & al.	sequential & parallel composition	parallel threads small step mix strict/lazy
M. Hidalgo-Herrero & Y. Ortega-Mallén	**Eden** process creation, dynamic channels, no determinism, streams	several processes small step eager/lazy

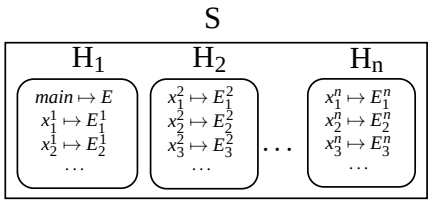


FIGURE 8.1. A distributed model

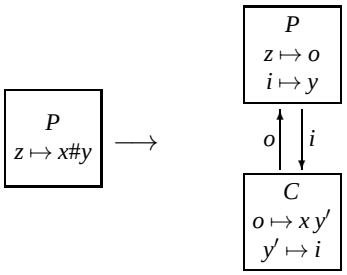


FIGURE 8.2. Process creation

The operational semantics for Eden of [6, 5] is inspired by an operational semantics for GpH [14] presented in [2]. In that work, labelled heaps are introduced to consider bindings as parallel threads. They also have two kinds of rules: single thread and multi-thread transitions, that relate roughly to our local and global transitions, respectively. In GPH there is no notion of independent process, and parallelism is reduced to sparkling threads in a common memory. This implies that there are not communications either. Besides, it is not necessary to copy bindings from one process to another. Hence, their model is much simpler than ours. Correctness, adequacy and determinacy are proved for the operational semantics of GPH in [2]. We are much indebted to this work: borrowing the main ideas that are used for their proofs, and adapting them to our case. Our task is not only complicated by the distributed memory, but also by the eagerness in creating processes and evaluating communications.

In order to favor parallelism, Eden overrides Haskell's laziness by creating process eagerly and by forcing the evaluation of communications. The mixture of laziness and strictness has been also investigated in the context of Launchbury's semantics; in [12], van Eekelen and de Mol extend the (sequential) calculus with a strict let-expression and prove the correctness and adequacy of their *mixed* semantics.

8.3 DISTRIBUTED MODEL

Our semantic model has to reflect two features: distribution and laziness, or more exactly, call by need. Computation is distributed among separate processes with no shared memory between them, although closures are shared inside each process. This leads to a two-level structure (see Figure 8.1): at the top-level is a distributed system (S) with parallel processes; each process is represented at the lower-level by a heap (H_i) of bindings from variables to expressions. Heaps represent the state of the computation in each process, where bindings correspond to potential concurrent threads. Thus, inside each process there is a kind of local parallelism with shared memory.

8.3.1 The Language

We extend the calculus used in [9] (an untyped lambda calculus extended with recursive local declarations) with a *parallel application* that leads to process cre-

ation. The abstract syntax is as follows:

$$E ::= x \mid \lambda x.E \mid x\,y \mid x\#y \mid \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ x$$

where $x, y \in \mathit{Var}$ is the set of variables, and $E \in \mathit{EExp}$ the *extended expressions*.

The evaluation of a parallel application $x\#y$ inside a process P (the parent) triggers the creation of a new process C (the child), and of two new channels communicating P and C (i and o). The parent process sends to the child the value for y via the input channel i, while the child process sends to its parent (through the output channel o) the result of the application $x\,y$. This is illustrated in Figure 8.2. There are differences with the operator par of GPH . First, par is just an indication of the possibility of generating parallelism, that can be ignored by the compiler, whereas # requires the creation of a new process even if there is only one processor for the whole computation. Second, par generates a parallel spark that shares memory with the rest of computing threads, whereas # creates a separate process with its own memory. Sharing between parent and child is lost, this way produce work duplication and requires some form of communication between processes. # is a simplification of the process creation operator in Eden, where several input/output channels of differente types (for instance, streams with possibly infinite sequence of data) are allowed.

Notice that our syntax is restricted, so that arguments for functional and for parallel application are variables, as the body of the let-construct. This facilitates the definition of the operational rules, and it can be easily achieved by introducing new let-bindings. We also require that all variables are distinct (apply α-conversion). This restriction allows us to forget about scopes, and no garbage-collection is needed. A similar normalisation process is described by Launchbury in [9], although he only requires variables for the second argument of applications. This is insufficient for our needs[2].

In the rest of the paper the following naming conventions are used: $x,\ y,\ z \in \mathit{Var}$, representing variables; $i, o, ch \in \mathit{Chan}$, channel identifiers to model communications between processes; $\theta,\ \tau \in \mathit{Var} \cup \mathit{Chan}$; $p,\ q \in \mathit{IdProc}$, process identifiers; and $W \in \mathit{Val}$, values. Expressions are evaluated to *weak head normal form* (*whnf*), therefore, *Val* contains the terms of the form $\lambda x.E$.

8.3.2 Operational Semantics

Similarly to the operational semantics presented in [2], we consider heaps consisting of *labelled bindings* of the form $\theta \overset{\alpha}{\mapsto} E$, where $\theta \in \mathit{Var} \cup \mathit{Chan}$, $E \in \mathit{EExp} \cup \mathit{Chan}$, and $\alpha \in \{A, I, B\}$ is the label representing its state. A labelled binding can be seen as a computation thread, which is *active* (A) when being evaluated, *blocked* (B) when waiting for the evaluation of another binding, and *inactive* (I) when it has not been demanded yet or it has been already evaluated.

A *labelled heap*, $H \in \mathit{LHeap}$, is a finite set of distinct named labelled bindings. A *process* is a pair of the form $\langle p, H \rangle$ where $p \in \mathit{IdProc}$ and H is a labelled heap.

[2] In fact, in [2] the evaluation of some apparently correct expressions gets blocked. This could be easily corrected by restricting more their syntax.

(value)	$H + \{x \overset{I}{\mapsto} W\} \,:\, \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} W\}$	
(demand)	$H + \{x \overset{IAB}{\mapsto} E\} \,:\, \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\}$	if $E \notin Val$
(blackhole)	$H \,:\, x \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{B}{\mapsto} x\}$	
(app. dem.)	$H + \{x \overset{IAB}{\mapsto} E\} \,:\, \theta \overset{A}{\mapsto} x\,y \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\,y\}$	if $E \notin Val$
(β-reduc.)	$H + \{x \overset{I}{\mapsto} \lambda z.E\} \,:\, \theta \overset{A}{\mapsto} x\,y \longrightarrow H + \{x \overset{I}{\mapsto} \lambda z.E, \theta \overset{A}{\mapsto} E[y/z]\}$	
(let)	$H \,:\, \theta \overset{A}{\mapsto} \texttt{let}\ \{x_i = E_i\}_{i=1}^n\ \texttt{in}\ x \longrightarrow$	
	$\longrightarrow H + \{y_i \overset{I}{\mapsto} E_i[y_j/x_j]_{j=1}^n\}_{i=1}^n + \{\theta \overset{A}{\mapsto} x[y_j/x_j]_{j=1}^n\}$	y_i *fresh*, $1 \le i \le n$

FIGURE 8.3. Local Rules

A *system S* is a nonempty set of processes. Initially a system consists of a unique heap with a unique (active) binding corresponding to the *main* expression. The system evolves until either a value is obtained for the *main* variable or the system gets blocked:

$$\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* \langle p_0, H + \{main \overset{I}{\mapsto} W\}\rangle, \langle p_1, H_1\rangle, \ldots, \langle p_n, H_n\rangle \Longrightarrow^*$$

where the transition $\Longrightarrow = \overset{par}{\Longrightarrow}; \overset{sys}{\Longrightarrow}$ involves *local* ($\overset{par}{\Longrightarrow}$) and *global* ($\overset{sys}{\Longrightarrow}$) rules. The former reflect the internal behavior of each process in the system, whereas the latter model the interaction between them.

Local rules. (See Figure 8.3) They model the evolution of an individual labelled heap, and they have the following general form:

$$\underbrace{\underbrace{H + \{\theta' \overset{\alpha}{\mapsto} E'\}}_{H''} : \theta \overset{A}{\mapsto} E}_{H_0} \longrightarrow H', \quad \text{with } \theta, \theta' \in Var \cup Chan$$

where the initial heap H_0 is modified to H'. The active binding $\theta \overset{A}{\mapsto} E$ represents the trigger of the evaluation. In some rules, another binding $\theta' \overset{\alpha}{\mapsto} E'$ from the rest of the heap, H'', plays a role in the evaluation of $\theta \overset{A}{\mapsto} E$. Occasionally, we use multi-labelled bindings to avoid rewriting rules that only differ on the label of some binding.

The rules describe the flow of demand until a *whnf* is reached. Notice that the rule *let* introduces new bindings. These are inactive because they have not been demanded yet. To avoid name clashes the variables are freshly renamed.

The possibility of evaluating simultaneously several active bindings is restricted by the number of available processors, but in our case we assume they are unlimited. For each heap H we can define a set of *evolutionary bindings*, $EB(H)$, containing those active threads in H that are allowed to evolve. The following rule describes how a process evolves by applying to each evolutionary binding in its heap a suitable local rule:

local parallel

$$\frac{\{H^{(i,1)} + H^{(i,2)} : \theta^i \overset{A}{\mapsto} E^i \longrightarrow H^{(i,1)} + K^{(i,2)} | H = H^{(i,1)} + H^{(i,2)} + \{\theta^i \overset{A}{\mapsto} E^i\} \wedge \theta^i \overset{A}{\mapsto} E^i \in EB(H)\}_{i=1}^n}{H \overset{lpar}{\longrightarrow} (\bigcap_{i=1}^n H^{(i,1)}) \bigcup (\bigcup_{i=1}^n K^{(i,2)})}$$

(unblocking) $(S, \langle p, H + \{x \overset{A}{\mapsto} W, \theta \overset{B}{\mapsto} E^x_B\}\rangle) \overset{unbl}{\to} (S, \langle p, H + \{x \overset{A}{\mapsto} W, \theta \overset{A}{\mapsto} E^x_B\}\rangle)$

(deactivation) $(S, \langle p, H + \{\theta \overset{A}{\mapsto} W\}\rangle) \overset{deact}{\to} (S, \langle p, H + \{\theta \overset{I}{\mapsto} W\}\rangle)$

(blocking p. c.) $(S, \langle p, H + \{\theta \overset{IA}{\mapsto} x\#y\}\rangle) \overset{bpc}{\to} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x\#y\}\rangle)$

(proc. creat.) if $\neg \mathsf{d}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\})$, q, z, i, o fresh, η fresh renaming

$$(S, \langle p, H + \theta \overset{\alpha}{\mapsto} x\#y\rangle) \overset{pc}{\to}$$
$$\overset{pc}{\to} (S, \langle p, H + \{\theta \overset{B}{\mapsto} o, i \overset{A}{\mapsto} y\}\rangle, \langle q, \eta(\mathsf{nh}(x, H)) + \{o \overset{A}{\mapsto} \eta(x)\ z, z \overset{B}{\mapsto} i\}\rangle)$$

(value comm.) if $\neg \mathsf{d}(W, H_p)$, η fresh renaming

$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{com}{\to}$$
$$\overset{com}{\to} (S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathsf{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} \eta(W)\}\rangle)$$

FIGURE 8.4. Global Rules

where $n = |EB(H)|$, that is, the number of evolutionary bindings in the heap H. $H^{(i,1)}$ represents the bindings of H that remain unchanged, while $H^{(i,2)}$ represents the bindings of H that are modified when the local rule is applied, and it becomes $K^{(i,2)}$. The rule is well-defined because it can be proved that the order in which the local rules are applied is irrelevant [5].

The exact definition of $EB(H)$ depends on the semantics that we desire to express. For instance, we can regulate the amount of speculative parallelism. For a *minimal semantics*, where there is no speculative computation, only the bindings that have been demanded by the main expression do evolve. Alternatively, to allow speculative computation (*maximal semantics*), the set of evolutionary bindings coincides with the set of active threads for each process.

The global rule *parallel* describes how the whole system evolves by applying the *local parallel* rule to each process in the system:

$$\textbf{(parallel)} \quad \frac{\{H_p \overset{lpar}{\longrightarrow} H'_p\}_{\langle p, H_p\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p, H_p\rangle \in S}}$$

Global Rules. (See Figure 8.4) They describe process creation and communication. They also take care of unblocking and deactivating the bindings.

Processes are created as soon as possible, thus introducing speculative parallelism. The rule *process creation* describes the changes produced in the system by the execution of a parallel application expression $x\#y$, i.e. it produces the structure described in Figure 8.2. Process q is created with an initial heap containing the bindings needed to compute x, i.e. $\mathsf{nh}(x, H)$. A fresh renaming is applied to this *needed heap* to avoid name clashing. Channels (i, o) are evaluated eagerly, hence they are created in an active state. A process is created only if this is *feasible*, i.e. if x has neither pending communications nor pending process creations in the needed heap for x. The definition of the auxiliary functions *needed heap* (nh) and *dependency* (d) are in Figure 8.5, where the function subexp returns the subexpressions of a given expression. If a parallel application has dependencies

$$\mathsf{d}(x,H) = \begin{cases} \mathit{true} & \text{if } \theta \overset{\alpha}{\mapsto} E \in \mathsf{nh}(x,H) \text{ where } (\theta \in \mathit{Var} \cup \mathit{Chan}) \wedge (E = \mathit{ch} \vee E = y\#z) \\ \mathit{false} & \text{in any other case} \end{cases}$$

$$\mathsf{nh}(E,H) \models \{\text{(i),(ii)}\} \wedge \forall A.(A \subseteq H \wedge A \models \{\text{(i),(ii)}\} \Rightarrow \mathsf{nh}(E,H) \subseteq A)$$

(i) $x \overset{\alpha}{\mapsto} E \in H \Rightarrow \{x \overset{I}{\mapsto} E\} \cup \mathsf{nh}(E,H) \subseteq \mathsf{nh}(x,H)$

(ii) $\bigcup_{E' \in \mathsf{subexp}(E)} \mathsf{nh}(E',H) \subseteq \mathsf{nh}(E,H)$

FIGURE 8.5. Dependency and needed heap functions

then the rule *blocking process creation* is applied.

Similarly, the rule *value communication* can only be aplied when the value to be communicated does not depend on another communication or process creation. The needed heap for the value is copied in the receiver. Once the communication has been completed, the corresponding channel is deleted.

An *expression blocked in the variable x*, E^x_B, is one of the following: x or $x\,y$. It indicates that the value of x is needed to continue with the evaluation of the blocked expression. Hence, if x is already bound to a value, then any expression blocked in x must be activated (rule *unblocking*).

If an expression has achieved a value it becomes inactive (rule *deactivation*).

The transition $\overset{sys}{\Longrightarrow} = \overset{com}{\Longrightarrow}; \overset{pc}{\Longrightarrow}; \overset{unbl}{\Longrightarrow}; \overset{deact}{\Longrightarrow}; \overset{bpc}{\Longrightarrow}$ collects the global rules. $\overset{com}{\Longrightarrow} = \overset{com}{\longrightarrow}^*$ means that the *value communication* rule is applied repeatedly until no more communications are possible; $\overset{pc}{\Longrightarrow}$, $\overset{unbl}{\Longrightarrow}$, $\overset{deact}{\Longrightarrow}$ and $\overset{bpc}{\Longrightarrow}$ are defined similarly.

8.4 PROPERTIES

Launchbury proved in [9] the correctness and adequacy of his big step natural semantics (NS) with respect to Abramsky's denotational semantics. In the present work we prove the same properties of the small step semantics introduced in Section 8.3.2 (nPS) with respect to an extended denotational semantics (EDS). To take advantage of Launchbury's work, we extend the NS with the parallel application (ENS). This extension is proved to be *consistent* with the NS. In order to relate the big step ENS with our small-step nPS we defined an intermediate semantics corresponding to the restriction of nPS to the case where there is only one available processor (1PS). It is shown that nPS and 1PS are equivalent (*determinacy*) and that 1PS and ENS are *equivalent*. We finish by proving the correctness and adequacy of the ENS with respect to the EDS. This is summarized in the diagram of Figure 8.6.

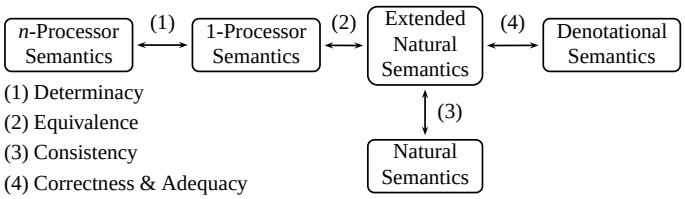


FIGURE 8.6. Semantics Equivalences Scheme

8.4.1 Denotational Semantics

If we just observe the final value produced for the main expression, then we can prove that our semantics is equivalent to a standard denotational semantics [1] where the denotation for parallel application is the same as for functional application. This means that speculative parallelism is semantically transparent, and it only affects the efficiency of the computation. As we have mentioned before, the difference between the minimal and the maximal semantics that we have defined above is the amount of speculative work. Hence, the maximal semantics may create more processes and produce more communications, but the value obtained for the main expresion is the same as with the minimal semantics.

We extend Abramsky's denotational semantics [1] for extended expressions and channels, so that the semantic function is :

$$[\![—]\!]_\rho : EExp \cup Chan \to Env \to Value$$

where the environment $\rho \in Env = Var \cup Chan \mapsto Value$ maps identifiers to values. The denotation of a parallel application agrees with the denotation of application, i.e. $[\![x\#y]\!]_\rho = [\![x\ y]\!]_\rho$, as the obtained final value is the same. The values are in the domain $Value = (Value \to Value)_\perp$.

8.4.2 1-Processor Semantics

In order to facilitate the task of relating our small-step semantics of section 8.3.1 (nPS) with Launchbury's big-step semantics, we restrict the nPS to the case where there is only one processor available (1PS). For this purpose, we impose an order in the evaluation of parallel threads, which is compatible with the minimal semantics. As a consequence, at each step of the computation, there will be at most one active binding. Thanks to fresh renaming when creating processes and communicating values, heaps associated with different processes are always disjoint. Therefore, we can consider a unique heap collecting all the bindings, instead of a system formed by separated heaps and we can avoid having rules at two levels.

Although there is no true parallelism in the case of one processor, we still create processes as soon as possible, as expressed by the rule *eager process creation*:

$$H : \theta \overset{I}{\mapsto} x\#y \overset{EPC}{\longrightarrow}_1 H + \{\theta \overset{I}{\mapsto} o,\ i \overset{I}{\mapsto} y,\ o \overset{I}{\mapsto} \eta(x)\ z,\ z \overset{I}{\mapsto} i\} + \eta(\mathsf{nh}(x,H))$$
$$\text{if } \neg\mathsf{d}(x, H + \{\theta \overset{I}{\mapsto} x\#y\}),\ \text{where } o,i,z \text{ are fresh, and } \eta \text{ is a fresh renaming}$$

(value) $H + \{x \overset{I}{\mapsto} W\} \ : \ \theta \overset{A}{\mapsto} x \ \longrightarrow_1 \ H + \{x \overset{I}{\mapsto} W,\ \theta \overset{A}{\mapsto} W\}$

(demand) if $E \notin Val$
$H + \{\tau \overset{IB}{\mapsto} E\} \ : \ \theta \overset{A}{\mapsto} \tau \ \longrightarrow_1 \ H + \{\tau \overset{AB}{\mapsto} E,\ \theta \overset{B(\tau)}{\mapsto} \tau\}$

(blocking I) $H \ : \ x \overset{A}{\mapsto} x \ \longrightarrow_1 \ H + \{x \overset{B(x)}{\mapsto} x\}$

(app. demand) if $E \notin Val$
$H + \{x \overset{IB}{\mapsto} E\} \ : \ \theta \overset{A}{\mapsto} x\,y \ \longrightarrow_1 \ H + \{x \overset{AB}{\mapsto} E,\ \theta \overset{B(x)}{\mapsto} x\,y\}$

(β-reduction) $H + \{x \overset{I}{\mapsto} \lambda z.E\} \ : \ \theta \overset{A}{\mapsto} x\,y \ \longrightarrow_1 \ H + \{x \overset{I}{\mapsto} \lambda z.E,\ \theta \overset{A}{\mapsto} E[y/z]\}$

(let) y_i fresh, $1 \le i \le n$
$H : \ \theta \overset{A}{\mapsto} \texttt{let}\ \{x_i = E_i\}_{i=1}^n\ \texttt{in}\ x \longrightarrow_1 \ H + \{y_i \overset{I}{\mapsto} E_i[y_j/x_j]_{j=1}^n\}_{i=1}^n + \{\theta \overset{A}{\mapsto} x[y_j/x_j]_{j=1}^n\}$

(unbl-deact) $H + \{\theta \overset{B(\tau)}{\mapsto} E\} : \tau \overset{A}{\mapsto} W \longrightarrow_1 \ H + \{\tau \overset{I}{\mapsto} W,\ \theta \overset{A}{\mapsto} E\}$

(proc. creat.) if $\neg \mathsf{d}(x, H^*)$, o, i, z, η fresh where $H^* = H + \{\theta \overset{A}{\mapsto} x\#y\}$
$H : \theta \overset{A}{\mapsto} x\#y \longrightarrow_1 \ H + \{\theta \overset{A}{\mapsto} o,\ i \overset{I}{\mapsto} y,\ o \overset{I}{\mapsto} \eta(x)\ z,\ z \overset{I}{\mapsto} i\} + \eta(\mathsf{nh}(x, H))$

(value comm.) if $\neg \mathsf{d}(W, H)$, η fresh renaming
$H + \{ch \overset{I}{\mapsto} W\} \ : \ \theta \overset{A}{\mapsto} ch \ \longrightarrow_1 \ H + \{\theta \overset{A}{\mapsto} \eta(W)\} + \eta(\mathsf{nh}(W, H))$

(blocking II) if $\mathsf{d}(x, H'') \wedge \mathsf{fd}(x, H'') = \theta \overset{A}{\mapsto} x\#y$ where $H'' = H + \{\theta \overset{A}{\mapsto} x\#y\}$
$H \ : \ \theta \overset{A}{\mapsto} x\#y \ \longrightarrow_1 \ H + \{\theta \overset{B(\theta)}{\mapsto} x\#y\}$

(pc. demand) if $\mathsf{d}(x, H'') \wedge \mathsf{fd}(x, H'') = z \overset{IB}{\mapsto} E$ where $H'' = H + \{z \overset{IB}{\mapsto} E, \theta \overset{A}{\mapsto} x\#y\}$
$H + \{z \overset{IB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\#y \longrightarrow_1 \ H + \{z \overset{AB}{\mapsto} E,\ \theta \overset{B(z)}{\mapsto} x\#y\}$

(comm. demand) if $\mathsf{d}(W, H'') \wedge \mathsf{fd}(W, H'') = z \overset{IB}{\mapsto} E$ where $H'' = H + \{z \overset{IB}{\mapsto} E\}$
$H + \{z \overset{IB}{\mapsto} E, ch \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} ch \longrightarrow_1 \ H + \{z \overset{AB}{\mapsto} E,\ ch \overset{I}{\mapsto} W, \theta \overset{B(z)}{\mapsto} ch\}$

FIGURE 8.7. 1-processor rules

The (unique) heap is modified by adding a fresh renaming of the needed heap for the process abstraction, but the active binding does not evolve, and no new binding is activated. As before, only feasible process creations can be completed.

Figure 8.7 shows the rest of rules adapted to the 1-processor case, where the (unique) active binding triggers the evaluation. The labels for blocked bindings are decorated with a name indicating the cause of the blocking ($B(\theta)$ indicates that the value of θ is needed to continue with the evaluation). The first six rules in Figure 8.7 are derived from the local rules in Figure 8.3. It is ensured that there is at most one active binding at each time. As channels are now created inactive, the new rule *demand* is defined for channels too.

The global rules *unblocking* and *deactivation* (see Figure 8.4) are unified in the rule *unblocking-deactivation*. In the nPS, several bindings can be blocked in the same name; hence, two steps are needed: first, bindings are unblocked —as many as possible—, afterwards any binding associated to a value is deactivated. However, in the 1PS, at most one binding can be blocked in each name, so that we can unblock and deactivate in the same step.

The rule *process creation* is applied when a parallel application has been de-

$$
\begin{aligned}
\mathsf{lsubexp}(x) &= [\,] \\
\mathsf{lsubexp}(\lambda x.E) &= [E] \\
\mathsf{lsubexp}(E_1\ E_2) &= [E_1, E_2] \\
\mathsf{lsubexp}(E_1 \# E_2) &= [E_1, E_2] \\
\mathsf{lsubexp}(\mathtt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \mathtt{in}\ E) &= [E_1,\ \ldots,\ E_n, E]
\end{aligned}
$$

$$
\mathsf{fd}(x,H) = \begin{cases} x \overset{\alpha}{\mapsto} E & \text{if } x \overset{\alpha}{\mapsto} E \in H \wedge (E \equiv ch \vee E \equiv y\#z) \\ \mathsf{fd}(E,H) & \text{if } x \overset{\alpha}{\mapsto} E \in H \wedge E \not\equiv ch \wedge E \not\equiv y\#z \\ \bot & \text{in any other case} \end{cases}
$$

$$
\mathsf{fd}(E,H) = \mathsf{fb}(\mathsf{lsubexp}(E), H) \quad \text{if } E \notin \mathit{Var}
$$

$$
\mathsf{fb}([\,],H) = \bot
$$

$$
\mathsf{fb}(E : L, H) = \begin{cases} \mathsf{fd}(E,H) & \text{if } \mathsf{fd}(E,H) \neq \bot \\ \mathsf{fb}(L,H) & \text{if } \mathsf{fd}(E,H) = \bot \end{cases}
$$

FIGURE 8.8. First dependency function

manded and it has no dependencies. Only the binding that has demanded the parallel application remains active, and this is bound to a new channel.

The rule *communication* can be applied if the channel has been demanded and the obtained value has no dependencies.

The rule *process creation demand* (*communication demand*) blocks a process creation (resp. a communication) that has dependencies. Only the first dependency —calculated by the *first dependency* function, fd (Figure 8.8)— is activated. Function fd is defined in terms of the *list of dependency subexpresions* function, lsubexp, and the *first binding* function, fb. The former returns the list of subexpresions of a given expresion, while the latter returns the first dependency binding $x \overset{\alpha}{\mapsto} E$ (with $E \equiv ch$ or $E \equiv y\#z$) of a given list of expressions and a given heap. $\equiv$ stands for the syntactic equivalence.

The *blocking* rules reflect blackholes.

The resulting transition relation has the form: $\Longrightarrow_1 = \longrightarrow_1 ; \overset{EPC}{\Longrightarrow}_1$. A transition begins by applying one of the rules of Figure 8.7, and then the *eager process creation* rule until all the feasible process creations have been completed, i.e. $\overset{EPC}{\Longrightarrow}_1 = \overset{EPC}{\longrightarrow}^*$.

Determinacy

The operational semantics defined in Section 8.3.1 is *determinant* in the sense that the obtained final value is independent of the number of processors available. This property of *determinacy* can be derived from the fact that the nPS and the 1PS (Section 8.4.2) compute the same values.

To prove this result it is necessary to construct a (unique) labelled heap from a process system and conversely, to recover the process system from a unique labelled heap . This can be easily achieved by adding a process identifier to each

Let $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$, $\theta \notin dom(\overline{\Gamma}), E \in EExp \cup Chan$ and $\Gamma^+ = \Gamma \cup \Gamma^B \cup \{\theta \mapsto E\}$

$\mathsf{sat}(\overline{\Gamma} : \theta \mapsto E) = \langle \Delta, \Gamma^B \rangle$ where Δ is the greatest heap that satisfies (i)-(iii)

(i) $\tau \mapsto x\#y \in \Gamma \wedge \mathsf{d}(x, \Gamma^+) \Rightarrow \tau \mapsto x\#y \in \Delta$

(ii) $\tau \mapsto x\#y \in \Gamma \wedge \neg\mathsf{d}(x, \Gamma^+) \Rightarrow \{\tau \mapsto o, i \mapsto y, o \mapsto \eta(x)\ z, z \mapsto i\} \cup \eta(\mathsf{nh}(x, \Gamma^+)) \subseteq \Delta$
o, i, z fresh names, η fresh renaming

(iii) $\tau \mapsto E \in \Gamma$ with $(E \not\equiv x\#y) \Rightarrow \tau \mapsto E \in \Delta$

FIGURE 8.9. Saturation function

labelled binding $(\theta \overset{\alpha}{\mapsto}_p E)$ in the 1PS. This identifier is transparent for most of the rules given for the 1PS, although care must be taken to put the correct process identifier when adding new bindings .

The determinacy can then be established by proving that for any $E \in EExp$ if $\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* S \Longrightarrow^* S'$, then there exists a computation $\{main \overset{A}{\mapsto} E\} \Longrightarrow_1^* H_S \Longrightarrow_1^* H_{S'}$, where H_S (resp. $H_{S'}$) is the heap constructed from S (resp. S'). And conversely, if $\{main \overset{A}{\mapsto} E\} \Longrightarrow_1^* H \Longrightarrow_1^* H'$, then there exists a computation $\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* S_H \Longrightarrow^* S_{H'}$, where S_H (resp. $S_{H'}$) is the process system recovered from H (resp. H').

8.4.3 Extended Natural Semantics

We extend the NS given in [9] with the *parallel application*. In the NS bindings are unlabelled and there is no notion of blocked threads. In fact, during the derivation some bindings —corresponding to blocked evaluations— disappear from the heap. However, blocked bindings are needed to determine whether a process creation is feasible or not. Therefore, we extend Launchbury's heaps with the set of bindings that have been blocked during a derivation (*heap of blockades*). As in the 1PS defined in the previous section blocked bindings are annotated with a label indicating in which variable or channel the binding has been blocked. An *extended heap* ($\overline{\Gamma}$) is a pair $\langle \Gamma, \Gamma^B \rangle$, where Γ is a labelled heap and Γ^B is a heap of blockades, and verifying that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma^B) = \emptyset$.

In the rules of the extended natural semantics (Figures 8.10 and 8.11), the expression that is being evaluated is bound to a name, so that assertions have the form $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, where $\theta \notin \mathsf{dom}(\overline{\Gamma}) \cup \mathsf{dom}(\overline{\Delta})$.

We use the notation $\overline{\Gamma} + \{\theta \overset{\alpha}{\mapsto} E\}$ to indicate that a new binding is added to an extended heap $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$. The label α can be either $B(\tau)$ or nothing. In the first case the binding is added to Γ^B, in the other case to Γ.

As we have explained before, processes are created as soon as possible. Therefore, extended heaps are *saturated*, i.e., there are no pending feasible process creations. A process creation is feasible in a extended heap $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$ if it has dependencies neither in the heap $\overline{\Gamma}$, nor in the name that is being evaluated. It is easy to prove that the order in which process creations are completed when saturating an extended heap is irrelevant. The saturation function (Figure 8.9) is equivalent to the EPC rule of the 1PS. To simplify the presentation of the rules we

(Lambda) $\mathsf{sat}(\overline{\Gamma} : \theta \mapsto \lambda x.E) \Downarrow \mathsf{sat}(\overline{\Gamma} : \theta \mapsto \lambda x.E)$ if $\theta \notin dom(\overline{\Gamma})$

(Applicat.)
$$\frac{\overline{\Gamma} + \{\theta \overset{B(x)}{\mapsto} x\,y\} : x \mapsto E \Downarrow \overline{\Delta} + \{\theta \overset{B(x)}{\mapsto} x\,y\} : x \mapsto \lambda z.E' \quad \overline{\Delta} + \{x \mapsto \lambda z.E'\} : \theta \mapsto E'[y/z] \Downarrow \overline{\Theta} : \theta \mapsto W}{\overline{\Gamma} + \{x \mapsto E\} : \theta \mapsto x\,y \Downarrow \overline{\Theta} : \theta \mapsto W}$$

(Variable)
$$\frac{\overline{\Gamma} + \{\theta \overset{B(x)}{\mapsto} x\} : x \mapsto E \Downarrow \overline{\Delta} + \{\theta \overset{B(x)}{\mapsto} x\} : x \mapsto W}{\overline{\Gamma} + \{x \mapsto E\} : \theta \mapsto x \Downarrow \overline{\Delta} + \{x \mapsto W\} : \theta \mapsto \widehat{W}}$$

(Let)
$$\frac{\mathsf{sat}(\overline{\Gamma} + \{y_i \mapsto E_i[y_j/x_j]_{j=1}^n\}_{i=1}^n : \theta \mapsto x[y_j/x_j]_{j=1}^n) \Downarrow \overline{\Delta} : \theta \mapsto W}{\overline{\Gamma} : \theta \mapsto \mathtt{let}\ \{x_i = E_i\}_{i=1}^n\ \mathtt{in}\ x \Downarrow \overline{\Delta} : \theta \mapsto W}$$

FIGURE 8.10. Extended natural semantics

(Process creation)
$$\frac{\Gamma' = \Gamma \cup \Gamma^B \cup \{\theta \mapsto x\#y\},\ \text{if } \neg\mathsf{d}(x,\Gamma') \quad \overline{\Gamma} + \{i \mapsto y, o \mapsto \eta(x)\,z, z \mapsto i\} + \eta(\mathsf{nh}(x,\Gamma')) : \theta \mapsto o \Downarrow \overline{\Delta} : \theta \mapsto W}{\overline{\Gamma} : \theta \mapsto x\#y \Downarrow \overline{\Delta} : \theta \mapsto W}$$

(Proc. creat. dem.)
$$\frac{\Gamma' = \Gamma \cup \Gamma^B \cup \{z \mapsto E', \theta \mapsto x\#y\},\ \text{if } \mathsf{d}(x,\Gamma') \wedge \mathsf{fd}(x,\Gamma') = z \mapsto E' \wedge z \neq \theta \quad \overline{\Gamma} + \{\theta \overset{B(z)}{\mapsto} x\#y\} : z \mapsto E' \Downarrow \overline{\Delta} + \{\theta \overset{B(z)}{\mapsto} x\#y\} : z \mapsto W' \quad \overline{\Delta} + \{z \mapsto W'\} : \theta \mapsto x\#y \Downarrow \overline{\Theta} : \theta \mapsto W}{\overline{\Gamma} + \{z \mapsto E'\} : \theta \mapsto x\#y \Downarrow \overline{\Theta} : \theta \mapsto W}$$

(Communication)
$$\frac{\text{if } \neg\mathsf{d}(W,\overline{\Delta}) \quad \overline{\Gamma} + \{\theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto E \Downarrow \overline{\Delta} + \{\theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto W}{\overline{\Gamma} + \{ch \mapsto E\} : \theta \mapsto ch \Downarrow \mathsf{sat}(\overline{\Delta} \cup \eta(\mathsf{nh}(W,\overline{\Delta})) : \theta \mapsto \eta(W))}$$

(Comm. demand)
$$\frac{\Delta' = \Delta \cup \Delta^B \cup \{z \mapsto E'\},\ \text{if } \mathsf{d}(W,\Delta') \wedge \mathsf{fd}(W,\Delta') = z \mapsto E' \quad \overline{\Gamma} + \{\theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto E \Downarrow \overline{\Delta} + \{z \mapsto E', \theta \overset{B(ch)}{\mapsto} ch\} : ch \mapsto W \quad \overline{\Delta} + \{ch \mapsto W, \theta \overset{B(z)}{\mapsto} ch\} : z \mapsto E' \Downarrow \overline{\Theta} + \{ch \mapsto W, \theta \overset{B(z)}{\mapsto} ch\} : z \mapsto W' \quad \overline{\Theta} + \{z \mapsto W', ch \mapsto W\} : \theta \mapsto ch \Downarrow \overline{\Lambda} : \theta \mapsto W''}{\overline{\Gamma} + \{ch \mapsto E\} : \theta \mapsto ch \Downarrow \overline{\Lambda} : \theta \mapsto W''}$$

where $z \in Var$ and $o, i, \in Chan$ are fresh names, and η is a fresh renaming

FIGURE 8.11. Extended natural semantics: process creation and communication

abuse notation and write $\mathsf{sat}(\overline{\Gamma} : \theta \mapsto E)$ for $\mathsf{sat}(\overline{\Gamma} : \theta \mapsto E) : \theta \mapsto E$.

The rules for our extended natural semantics are similar to those given in [9], but including saturation, blocked bindings and names for the expression to be evaluated (Figures 8.10 and 8.11). We describe only the additional rules that describe process creation and communication (see Figure 8.11). The rule *Process creation* corresponds to the case of a feasible process creation, while the rule *Process creation demand* deals with the case where dependencies (d) are detected: the first dependency (fd) is evaluated, and then the process creation is retried. The rules for communication calculate the value associated to a channel. Demand is generated if the value to be communicated has dependencies.

Consistency

The ENS is *consistent* with the original NS, in the sense that, if we consider expressions without parallel application, then the same value is obtained in both derivation systems, and the final heaps can be related, i.e. for any $E \in Exp$ we have $\Gamma : E \Downarrow \Delta : W$ iff $\langle \Gamma, \Gamma^B \rangle : x \mapsto E \Downarrow \langle \Delta, \Gamma^B \rangle : x \mapsto W$ where Γ^B is disjoint from Γ and Δ, and x is completely a fresh variable.

Equivalence

The 1PS and the ENS are equivalent. There is an obvious conversion from labelled heaps to extended heaps (and viceversa), where inactive bindings are collected in the first component of $\overline{\Gamma}$ and blocked bindings in the second component. Therefore $H + \{\theta \overset{A}{\mapsto} E\} \Longrightarrow_1^* H' + \{\theta \overset{A}{\mapsto} W\}$ iff $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$ where $\overline{\Gamma}$ and $\overline{\Delta}$ are the extended heaps corresponding to H and H' respectively.

Correctness and Computational Adequacy

Following the steps given in [9], we extend the proofs for *correctness* and *computational adequacy* for the ENS respect to the EDS.

The correctness theorem establishes that the meaning of an expression does not change during its evaluation, and that heaps only change by adding new bindings or refining the closures of the existing bindings.

Theorem 8.1 (Correctness of the ENS). *Let $E \in EExp \cup Chan$, $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle$, $\overline{\Delta} = \langle \Delta, \Delta^B \rangle \in EHeap$, and $\theta \notin dom(\overline{\Gamma})$. If $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, then for every environment ρ, $[\![E]\!]_{\{\{\Gamma\}\}_\rho} = [\![W]\!]_{\{\{\Delta\}\}_\rho}$ and $\{\{\Gamma\}\}_\rho \leq \{\{\Delta\}\}_\rho$, where the semantic function: $\{\{\ldots\}\} : Heap \to Env \to Env$, defined as $\{\{x_1 \mapsto E_1, \ldots, x_n \mapsto E_n\}\}\rho = \mu\rho'.\rho \sqcup (x_1 \mapsto [\![E_1]\!]_{\rho'} \ldots x_n \mapsto [\![E_n]\!]_{\rho'})$, being μ the least fixed point operator, and Heap is the domain of (unlabelled) heaps, i.e. sets of unlabelled bindings. The ordering on environments is defined as in [9], so that $\rho \leq \rho'$ if ρ' binds more variables than ρ.*

The computational adequacy theorem states that an expression reduces to a value if and only if its denotation is non-bottom.

Theorem 8.2 (Adequacy: of the ENS). *Let $E \in EExp \cup Chan$, $\overline{\Gamma} = \langle \Gamma, \Gamma^B \rangle \in EHeap$, and $\theta \notin dom(\overline{\Gamma})$. $\overline{\Gamma} : \theta \mapsto E \Downarrow \overline{\Delta} : \theta \mapsto W$, iff $[\![E]\!]_{\{\{\Gamma\}\}_\rho} \neq \bot$.*

8.5 DISCUSSION

Proving semantic properties like correctness or computational adequacy is much more than just a theoretical exercise. The search for formal proofs of these properties has led to a deeper understanding of our distributed model and the interaction of distribution and parallelism with laziness. Moreover, it has revealed other interesting features such as how to sequentialize parallel computations and to establish an order in evaluation demand.

For instance, our extension of Launchbury's natural semantics could not be reduced to adding rules for dealing with the parallel application (the only extension in the syntax). It was necessary to add information about blocked bindings, and to *saturate* heaps to perform process creation eagerly.

8.6 FUTURE WORK

The syntax used in [6, 5] includes the parallel operator, and also communications streams, dynamic channels and non-determinism; an immediate future work is to extend the properties proved here to include these other features of Eden.

Although the semantic model presented here offers insights into the distribution of computation —the systems of processes that have been created, and the communications performed during evaluation can be observed—, for correctness, adequacy and determinacy, we have just considered the final value that is calculated. This relates to the *value* equivalence recently defined in [4]; in this work, three different notions of equivalence between expressions are defined. Roughly, two expressions are *value* equivalent if when they are evaluated on the same heap, the same value is obtained; they are *heap* equivalent if the same final heap is produced (the value obtained may differ); and they are *strict* equivalent if they coincide on the value and on the final heap. It would be interesting to investigate these equivalences in our semantics, and to prove elementary properties for our parallel application as those given for `seq` in [4], i.e. idempotence, left-commutativity and associativity.

We are also interested in extending our results for properties beyond functionality, like for instance work duplication, task distribution or value communication, that are meaningful in a parallel and distributed context. To this purpose, the process system has to become part of the denotation of a program, so that process creation and communication/synchronization are side-effects produced by the evaluation of expressions. This can be formalized by defining denotational values with an additional parameter: the *continuation*, i.e., a state transformer which gathers the effect of the context where an expression is evaluated. There exists for Eden a continuation-based denotational semantics [7, 8, 5], and our ultimate goal is to prove the equivalence between this denotational semantics and the operational one given in [6, 5]. For this, we have to find a way to relate the mechanism of continuations with that of reduction for the operational semantics.

Acknowledgements We recognise the support of the following grants: MEC: TIN2006-15660-C02-01, TIN2006-15578-C02-01 and BES-2007-16823; CAM: S-0505/TIC-0407. We thank Greg Michaelson and the anonymous referees for his valuable comments.

REFERENCES

[1] S. Abramsky. *Research Topics in Functional Programming*, chapter 4: The lazy lambda calculus, pages 65–117. Ed. D. A. Turner. Addison Wesley, 1990.

[2] C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173, Montreal, Canada, September 2000.

[3] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM – the DistRibuted Eden Abstract Machine. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages, (IFL'97 selected papers)*, pages 250–269. LNCS 1467, Springer, 1997.

[4] M. J. Gabbay, S. H. Haeri, Y. Ortega-Mallén, and P. W. Trinder. Reasoning about selective strictness: operational equivalence, heaps and call-by-need evaluation, new inductive principles. In *submitted to ACM-SIGPLAN International Conference on Functional Programming (ICFP'09)*, Edinburgh, UK, August 2009.

[5] M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo*. PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[6] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[7] M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *Proc. of the 1st Asian Symposium on Programming Languages and Systems*, pages 303–321. LNCS 2895, Springer, 2003.

[8] M. Hidalgo-Herrero and Y. Ortega-Mallén. Dealing denotationally with stream-based communication. *Electronic Notes in Theoretical Computer Science*, 137(1):47–68, 2005.

[9] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.

[10] R. Loogen. *Research Directions in Parallel Functional Programming*, chapter 3: Programming Language Constructs, pages 63–92. Eds. K. Hammond and G. Michaelson. Springer, 1999.

[11] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[12] M. van Eekelen and M. de Mol. Mixed Lazy/Strict Graph Semantics. Technical report, Christian-Albrechts-Universität zu Kiel", Technical Report 0408, 245-260, 2004, Lübeck, Germany.

[13] L. Sánchez-Gil. Sobre la equivalencia entre semánticas operacionales y denotacionales para lenguajes funcionales paralelos. Master's thesis, Universidad Complutense de Madrid (Departamento de Sistemas Informáticos y Computación), 2008. `http://maude.sip.ucm.es/eden-semantics/` .

[14] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Funcional Programming*, 8(1):23–60, 1998.

[15] P. W. Trinder, H. W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4+5):469–510, 2003.

Apéndice A

Versiones extendidas

Incluimos dos informes técnicos (TR1 [SGHHOM12c] y TR2 [SGHHOM13]) que son las versiones extendidas de las publicaciones en *International Colloquium on Theoretical Aspects of Computing* (ICTAC'12) y *Perspectives of System Informatics* (PSI'14), respectivamente. De hecho, los títulos de estos artículos coinciden con los incluidos en el Capítulo 5: *A locally nameless representation for a natural semantics for lazy evaluation* y *The role of indirections in lazy natural semantics*. Estas versiones extendidas incluyen explicaciones más detalladas y todas las demostraciones, no sólo de los resultados más importantes sino también de todos los resultados auxiliares que han sido necesarios para realizarlas.

We include two technical reports, (TR1 [SGHHOM12c] *and* TR2 [SGHHOM13])*, in this Appendix. Since they are the extended versions of the publications in* International Colloquium on Theoretical Aspects of Computing *(ICTAC'12) and* Perspectives of System Informatics *(PSI'14), their titles coincide with the ones included in Chapter 5:* A locally nameless representation for a natural semantics for lazy evaluation *and* The role of indirections in lazy natural semantics*. These extended versions include detailed proofs of all the results.*

A locally nameless representation for a natural semantics for lazy evaluation

Technical Report 01/12

Lidia Sánchez-Gil[1], Mercedes Hidalgo-Herrero[2], and Yolanda Ortega-Mallén[3]

[1] Dpto. Sistemas Informáticos y Computación, Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain
[2] Dpto. Didáctica de las Matemáticas, Facultad de Educación, Universidad Complutense de Madrid, Spain
[3] Dpto. Sistemas Informáticos y Computación, Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain

Abstract. We propose a locally nameless representation for Launchbury's natural semantics for lazy evaluation. Names are reserved for free variables, while bound variable names are replaced by indices. This avoids the use of α-conversion and facilitates the identification of equivalent values in reduction proofs. We use cofinite quantification to express the semantic rules that introduce fresh names, but we prove that existential rules are admissible too. Moreover, we prove that the choice of names during the evaluation of a term is irrelevant as long as they are fresh enough.

1 Introduction

In the usual representation of the lambda-calculus, i.e., with variable names for free and bound variables, terms are identified up to α-conversion. This notation is suitable for explaining new concepts and for giving examples, while α-substitution together with Barendregt's variable convention [2] are freely used in informal reasoning. But the variable convention may lead to prove false (see [8]), and α-substitution is hard to implement in an automatic proof assistant. Therefore, other representations have been proposed to avoid names and α-conversion. For instance, the de Bruijn notation [5], where variable names are replaced by indices. However, this nameless notation is much less intuitive and quite cumbersome to use, as small modifications of a term may imply multiple shiftings of the indices. A compromise between the named representation and the de Bruijn notation is the locally nameless representation as presented in [4]. In this case, bound variable names are replaced by indices, while free variables keep their names. This mixed notation combines the advantages of both named and nameless representations. On the one hand, α-conversion is no longer needed and variable substitution is easily defined because there is no danger of name capture. On the other hand, terms are still readable and easy to manipulate.

We use a locally nameless representation to express Launchbury's natural semantics for lazy evaluation [6]. Our final purpose is to implement this natural

$$x \in Var$$
$$e \in Exp ::= \lambda x.e \mid (e\ x) \mid x \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e.$$

Fig. 1. Restricted *named* syntax of the extended λ-calculus

semantics in some proof assistant like Coq [3], and then to prove formally several properties of the semantics. The reduction rule for local declarations implies the introduction of fresh names. We use neither an existential nor a universal rule for this case. Instead, we opt for a cofinite rule as introduced by Aydemir et al. in [1]. Nevertheless, an *introduction lemma* is stated (and proved) which expresses that an existential rule is admissible too. Our locally nameless semantics is completed with a *regularity lemma* which ensures that every term and heap involved in a reduction proof are well-formed, and with a *renaming lemma* which indicates that the choice of names (free variables) is irrelevant as long as they are fresh enough. We have experienced the advantages of using cofinite rules when demonstrating these results.

In summary, the contributions of this paper are:

1. A locally nameless representation of the λ-calculus extended with recursive local declarations;
2. A locally nameless version of the inductive rules of Launchbury's natural semantics for lazy evaluation;
3. A new version of cofinite rules where the variables quantified in the premises do appear in the conclusion too; and
4. A formal proof of several properties of our reduction system like the regularity, the introduction and the renaming lemmas.

The paper is structured as follows: In Section 2 we present the locally nameless representation of the lambda calculus extended with recursive local declarations. The locally nameless translation of the natural semantics for lazy evaluation given in [6] is described in Section 3, together with the regularity, the introduction and the renaming lemmas. The proofs of these lemmas and other auxiliary results are detailed in the Appendix. In Section 4 we draw conclusions and outline our future work.

2 The locally nameless representation

The language described in [6] is a normalized lambda calculus extended with recursive local declarations. We reproduce the restricted syntax in Figure 1. Normalization is achieved in two steps. First an α-conversion is performed so that all bound variables have distinct names. In a second phase, it is ensured that arguments for applications are restricted to be variables. These static transformations make more explicit the sharing of closures and, thus, simplify the definition of the reduction rules.

We give the corresponding locally nameless representation by following the methodology summarized in [4]:

$$
\begin{array}{lll}
x \in \mathit{Id} & & i, j \in \mathbb{N} \\
v \in \mathit{Var} & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in \mathit{LNExp} & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

Fig. 2. Locally nameless syntax

1. Define the syntax of the extended λ-calculus in the locally nameless style.
2. Define the variable opening and variable closing operations.
3. Define the free variables and substitution functions, as well as the local closure predicate.
4. State and prove the properties of the operations on terms that are needed in the development to be carried out.

2.1 Locally nameless syntax

The locally nameless (restricted) syntax is shown in Figure 2. *Var* stands now for the set of *variables*, where it is distinguished between *bound variables* and *free variables*. The calculus includes two variable binders: λ-abstraction and `let`-expression. Since `let` declarations are multibinders, bound variables are represented with two natural numbers: the first number indicates to which binder of the term (either abstraction or `let`) the variable is bound, while the second refers to the position of the variable inside the binder (in the case of an abstraction this second number should be 0). In the following, we will represent a list like $\{t_i\}_{i=1}^{n}$ as $\overline{t}$, with length $|\overline{t}| = n$.

Example 1. Let $e \in \mathit{Exp}$ be the λ-expression given in the named representation

$$e \equiv \lambda z.\texttt{let}\ \{x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2\}\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in \mathit{LNExp}$ is:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \{\texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\}\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms. □

Application arguments are still restricted to variables, but the first phase of the normalization (described at the beginning of the section) is no longer needed.

2.2 Variable opening and variable closing

Variable opening and *variable closing* are the main operations to manipulate locally nameless terms. We extend the definitions given in [4] to the `let`-expression defined in Figure 2.[4]

[4] Multiple binders are defined in [4]. One corresponds to non recursive local declarations, and the other to mutually recursive expressions. Both constructions are treated as extensions, so that they are not completely developed.

$$
\begin{array}{lll}
\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) & = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases} \\
\{k \to \overline{x}\}(\texttt{fvar}\ x) & = \texttt{fvar}\ x \\
\{k \to \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \to \overline{x}\}\ t) \\
\{k \to \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \to \overline{x}\}\ t)\ (\{k \to \overline{x}\}\ v) \\
\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \to \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}\ t)
\end{array}
$$

where $\{k \to \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \to \overline{x}\}\,\cdot)\ \overline{t}$.

Fig. 3. Variable opening

In order to be able to explore the body of a binder construction (abstraction or `let`), one needs to replace the corresponding bound variables by fresh names. In the case of an abstraction $\texttt{abs}\ t$ the *variable opening operation* provides a (fresh) name to replace in t the bound variables referring to the outermost abstraction. Analogously, the opening of a `let`-term $\texttt{let}\ \overline{t}\ \texttt{in}\ t$ provides a list of distinct fresh names (as many as local declarations in $\overline{t}$) to replace the bound variables occurring in $\overline{t}$ and in the body t that refer to this particular declaration.

Variable opening is defined in terms of a recursive function $\{k \to \overline{x}\}t$ (Figure 3), where the number k represents the nesting level of the binder of interest, and $\overline{x}$ is a list of pairwise-distinct identifiers in Id. Since the level of the outermost binder is 0, variable opening is defined as:

$$t^{\overline{x}} = \{0 \to \overline{x}\}t.$$

Sometimes we are interested in applying the opening operation to a list of terms: $\overline{t}^{\overline{x}} = \texttt{List.map}\ (\cdot^{\overline{x}})\ \overline{t}$.

The last definition and those in Figure 3 include some operations on lists. We use an ML-like notation. For instance, $\texttt{List.nth}\ j\ \overline{x}$ represents the $(j+1)^{th}$ element of $\overline{x}$,[5] and $\texttt{List.map}\ f\ \overline{t}$ indicates that the function f is applied to every term in the list $\overline{t}$. In the rest of definitions we will use similar list operations.

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that all names in $\overline{x}$ are different). The definition of variable closing is based on a recursive function $\{k \leftarrow \overline{x}\}t$ (Figure 4), where k indicates again the level of nesting of binders.Whenever a free variable $\texttt{fvar}\ x$ is encountered, x is looked up in $\overline{x}$. If x occurs in position j, then the free variable is replaced by the bound variable $(\texttt{bvar}\ k\ j)$, otherwise it is left unchanged.Variable closing is then defined as follows:

$${}^{\backslash\overline{x}}t = \{0 \leftarrow \overline{x}\}t.$$

Variable closing of a list of terms is: ${}^{\backslash\overline{x}}\overline{t} = \texttt{List.map}\ ({}^{\backslash\overline{x}}\cdot)\ \overline{t}$.

[5] In order to better accommodate to bound variables indices, elements in a list are numbered starting with 0.

$$
\begin{array}{lll}
\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & = \texttt{bvar}\ i\ j & \\
\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) & = \begin{cases} \texttt{bvar}\ k\ j & \text{if } \exists j : 0 \leq j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise} \end{cases} & \\
\{k \leftarrow \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}\ t) & \\
\{k \leftarrow \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \leftarrow \overline{x}\}\ t)\ (\{k \leftarrow \overline{x}\}\ v) & \\
\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}\ t) &
\end{array}
$$

where $\{k \leftarrow \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \leftarrow \overline{x}\}\ \cdot)\ \overline{t}$.

Fig. 4. Variable closing

$$
\text{LC\_VAR}\ \frac{}{\texttt{lc}\ (\texttt{fvar}\ x)} \qquad \text{LC\_ABS}\ \frac{\forall x \notin L \subseteq Id \quad \texttt{lc}\ t^{[x]}}{\texttt{lc}\ (\texttt{abs}\ t)}
$$

$$
\text{LC\_APP}\ \frac{\texttt{lc}\ t \quad \texttt{lc}\ v}{\texttt{lc}\ (\texttt{app}\ t\ v)} \qquad \text{LC\_LET}\ \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id \quad \texttt{lc}\ [t : \overline{t}]^{\overline{x}}}{\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}
$$

$$
\text{LC\_LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc}\ \cdot)\ \overline{t}}{\texttt{lc}\ \overline{t}}
$$

Fig. 5. Local closure

2.3 Local closure, free variables and substitution

The locally nameless syntax in Figure 2 allows to build terms that have no corresponding expression in *LNExp* (Figure 1). For instance, the term `abs (bvar 1 5)` is an improper syntactic object, since index 1 does not refer to a binder in the term. The well-formed terms, i.e., those that correspond to expressions in *LNExp*, are called *locally closed*.

To determine if a term is locally closed one should check that any bound variable in the term has valid indices, i.e., that they refer to binders in the term. However, this checking is not straightforward, and an easier method is to open with fresh names every abstraction and `let`-expression in the term to be checked, and prove that no bound variable is ever reached. This checking is implemented with the *local closure* predicate `lc` t given in Figure 5.

Observe that cofinite quantification rules [1] are used for the binders, i.e., abstraction and `let`. Cofinite quantification is an elegant alternative to exist-fresh conditions and provides stronger induction and inversion principles. Proofs are simplified, because it is not required to define exactly the set of fresh names (several examples of this are given in [4]). The rule LC-ABS establishes that an abstraction is locally closed if there exists a finite set of names L such that, for any name x not in L, the term $t^{[x]}$ is locally closed. Similarly, the rule LC-LET indicates that a `let`-expression is locally closed if there exists a finite set of names L such that, for any list of distinct names $\overline{x}$ not in L and of length $|\overline{t}|$ ($\overline{x}^{|\overline{t}|} \notin L$), the opening of each term in the list of local declarations, $\overline{t}^{\overline{x}}$, and of the term affected by these declarations, $t^{\overline{x}}$, is locally closed. We use the notation $[t : \overline{t}]$ to represent the list with head t and tail $\overline{t}$. The empty list is represented as

$$\text{LCK-BVAR}\quad \frac{i < k \wedge j < \texttt{List.nth}\ i\ \overline{n}}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)} \qquad \text{LCK-APP}\quad \frac{\texttt{lc\_at}\ k\ \overline{n}\ t \qquad \texttt{lc\_at}\ k\ \overline{n}\ v}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t\ v)}$$

$$\text{LCK-FVAR}\quad \frac{}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{fvar}\ x)} \qquad \text{LCK-LET}\quad \frac{\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ [t : \overline{t}]}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LCK-ABS}\quad \frac{\texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ t}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t)} \qquad \text{LCK-LIST}\quad \frac{\texttt{List.forall}\ (\texttt{lc\_at}\ k\ \overline{n}\ \cdot)\ \overline{t}}{\texttt{lc\_at}\ k\ \overline{n}\ \overline{t}}$$

Fig. 6. Closed at level k

$[\,]$, a unitary list as $[t]$, and $[\overline{t} : t]$ stands for $\overline{t} +\!\!+ [t]$, where $+\!\!+$ is the concatenation of lists.

Coming back to the first approach to local closure, i.e., checking that indices in bound variables are valid, a new predicate is defined: *t is closed at level k*, written $\texttt{lc\_at}\ k\ \overline{n}\ t$ (Figure 6), where k indicates the current depth, that is, how many binders have been passed by. As binders can be either abstractions or local declarations, we need to keep the size of each binder (1 in case of an abstraction, n for a `let`-expression with n local declarations). These sizes are collected in the list $\overline{n}$, thus $|\overline{n}|$ should be at least k. A bound variable $\texttt{bvar}\ i\ j$ is closed at level k if i is smaller than k and j is smaller than $\texttt{List.nth}\ i\ \overline{n}$. The list $\overline{n}$ is new with respect to [4] because there the predicate `lc_at` is not defined for multiple binders.

We can define an order between lists of natural numbers as follows:

$$[\,] \geq [\,] \qquad\qquad m \geq n \wedge \overline{m} \geq \overline{n} \Rightarrow [m : \overline{m}] \geq [n : \overline{n}]$$

If a term t is locally closed at level k for a given list of numbers $\overline{n}$, then it is also locally closed at level k for any list of numbers greater than $\overline{n}$.

Lemma 1.
LC_AT_M_FROM_N $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \Rightarrow \forall \overline{m} \geq \overline{n}\,.\,\texttt{lc\_at}\ k\ \overline{m}\ t$

The two approaches are equivalent, so that it can be proved that a term is locally closed if and only if it is closed at level 0.

Lemma 2.
LC_IIF_LC_AT $\quad \texttt{lc}\ t \Leftrightarrow \texttt{lc\_at}\ 0\ [\,]\ t$

Computing the *free variables* of a term t is very easy in the locally nameless representation, since bound and free variables are syntactically different. The set of free variables of $t \in \mathit{LNExp}$ is denoted as $\texttt{fv}(t)$, and it is defined in Figure 7.

A name x is said to be *fresh for a term t*, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t:

$$\frac{x \notin \texttt{fv}(t)}{\texttt{fresh}\ x\ \texttt{in}\ t}$$

$$\begin{array}{lll} \mathtt{fv}(\mathtt{bvar}\ i\ j) & = \emptyset & \mathtt{fv}(\mathtt{fvar}\ x) = \{x\} \\ \mathtt{fv}(\mathtt{app}\ t\ v) & = \mathtt{fv}(t) \cup \mathtt{fv}(v) & \mathtt{fv}(\mathtt{abs}\ t) = \mathtt{fv}(t) \\ \mathtt{fv}(\mathtt{let}\ \overline{t}\ \mathtt{in}\ t) & = \mathtt{fv}(\overline{t}) \cup \mathtt{fv}(t) & \end{array}$$

where $\mathtt{fv}(\overline{t}) = \mathtt{List.foldright}\ (\cdot \cup \cdot)\ \emptyset\ (\mathtt{List.map}\ \mathtt{fv}\ \overline{t})$.

Fig. 7. Free variables

$$\begin{array}{lll} (\mathtt{bvar}\ i\ j)[z/y] & = \mathtt{bvar}\ i\ j & (\mathtt{fvar}\ x)[z/y] = \begin{cases} \mathtt{fvar}\ z & \text{if } x = y \\ \mathtt{fvar}\ x & \text{if } x \neq y \end{cases} \\ (\mathtt{abs}\ t)[z/y] & = \mathtt{abs}\ t[z/y] & (\mathtt{app}\ t\ v)[z/y] = \mathtt{app}\ t[z/y]\ v[z/y] \\ (\mathtt{let}\ \overline{t}\ \mathtt{in}\ t)[z/y] & = \mathtt{let}\ \overline{t}[z/y]\ \mathtt{in}\ t[z/y] & \end{array}$$

where $\overline{t}[z/y] = \mathtt{List.map}\ ([z/y]\cdot)\ \overline{t}$.

Fig. 8. Substitution

This definition can be easily extended to a list of distinct names $\overline{x}$:

$$\frac{\overline{x} \notin \mathtt{fv}(t)}{\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ t}$$

A term t is *closed* if it has no free variables at all:

$$\frac{\mathtt{fv}(t) = \emptyset}{\mathtt{closed}\ t}$$

Substitution replaces a variable name by another name in a term. So that for $t \in \mathit{LNExp}$ and $z, y \in \mathit{Id}$, $t[z/y]$ is the term where z substitutes any occurrence of y in t (see Figure 8).

Under some conditions variable closing and variable opening are inverse operations. More precisely, opening a term with fresh names and closing it with the same names, produces the original term. Symmetrically, closing a locally closed term t and then opening it with the same names gives back t.

Lemma 3.

CLOSE_OPEN_VAR	$\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ t \Rightarrow {}^{\backslash \overline{x}}(t^{\overline{x}}) = t$
OPEN_CLOSE_VAR	$\mathtt{lc}\ t \Rightarrow ({}^{\backslash \overline{x}}t)^{\overline{x}} = t$

3 Natural semantics for lazy λ-calculus

The natural semantics defined by Launchbury [6] follows a lazy strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : w$, that is, the expression $e \in \mathit{Exp}$ in the context of the heap Γ reduces to the value w in the context of the (modified) heap Δ. *Values* ($w \in \mathit{Val}$) are expressions in weak-head-normal-form (*whnf*). *Heaps* are partial functions from variables into expressions. Each pair (variable, expression) is called a *binding*, and it is represented by $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings may be updated to their

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : w}$$

Fig. 9. Natural semantics

corresponding computed values. The rules of this natural semantics are shown in Figure 9. The normalization of the λ-calculus, that has been mentioned in Section 2, simplifies the definition of the operational rules, although a renaming is still needed ($\hat{w}$ in VAR) to avoid name clashing. This renaming is justified by the Barendregt's variable convention [2].

3.1 Locally nameless heaps

Before translating the semantic rules in Figure 9 to the locally nameless representation defined in Section 2, we have to establish how *bindings* and *heaps* are represented in this notation.

Recall that bindings associate expressions to free variables, therefore bindings are now pairs $(\texttt{fvar}\ x, t)$ with $x \in Id$ and $t \in LNExp$. To simplify, we will just write $x \mapsto t$. In the following, we will represent a heap $\{x_i \mapsto t_i\}_{i=1}^n$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as $LNHeap$.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names that are bound in the heap.

$$\texttt{dom}(\emptyset) = \emptyset \qquad \texttt{dom}(\Gamma, x \mapsto t) = \texttt{dom}(\Gamma) \cup \{x\}$$

In a well-formed heap names are defined at most once and terms are locally closed. The predicate `ok` expresses that a heap is well-formed:

$$\textsc{ok-empty}\ \frac{}{\texttt{ok}\ \emptyset} \qquad \textsc{ok-cons}\ \frac{\texttt{ok}\ \Gamma \qquad x \notin \texttt{dom}(\Gamma) \qquad \texttt{lc}\ t}{\texttt{ok}\ (\Gamma, x \mapsto t)}$$

A similar (but related with normalization) predicate *distinctly named* is defined in [6] for heap/term pairs.

The function `names` returns the set of names that appear in a heap, i.e., the names occurring in the domain or in the right side terms:

$$\texttt{names}(\emptyset) = \emptyset \qquad \texttt{names}(\Gamma, x \mapsto t) = \texttt{names}(\Gamma) \cup \{x\} \cup \texttt{fv}(t)$$

This definition can be extended to the context of a heap/term pair:

$$\texttt{names}(\Gamma : t) = \texttt{names}(\Gamma) \cup \texttt{fv}(t)$$

We use it to define the freshness predicate of a list of names in a heap/term pair:

$$\text{LNLam}\quad \frac{\{\texttt{ok}\ \Gamma\} \qquad \{\texttt{lc}\ (\texttt{abs}\ t)\}}{\Gamma : \texttt{abs}\ t \Downarrow \Gamma : \texttt{abs}\ t}$$

$$\text{LNVar}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \cup \texttt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : (\texttt{fvar}\ x) \Downarrow (\Delta, x \mapsto w) : w}$$

$$\text{LNApp}\quad \frac{\Gamma : t \Downarrow \Theta : \texttt{abs}\ u \qquad \Theta : u^{[x]} \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin \texttt{dom}(\Delta)\}}{\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x) \Downarrow \Delta : w}$$

$$\text{LNLet}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \qquad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Fig. 10. Locally nameless natural semantics

$$\frac{\overline{x} \notin \texttt{names}(\Gamma : t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : t)}$$

Substitution of variable names is extended to heaps as follows:

$$\emptyset[z/y] = \emptyset \qquad (\Gamma, x \mapsto t)[z/y] = (\Gamma[z/y], x[z/y] \mapsto t[z/y])$$
$$\textit{where } x[z/y] = \begin{cases} z & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

The following property is verified:

Lemma 4.
OK_SUBS_OK $\quad \texttt{ok}\ \Gamma \wedge y \notin \texttt{dom}(\Gamma) \Rightarrow \texttt{ok}\ \Gamma[y/x]$

3.2 Locally nameless semantics

Once the locally nameless syntax and the corresponding operations, functions and predicates have been defined, three steps are sufficient to translate an inductive definition on λ-terms from the named representation into the locally nameless notation (as it is explained in [4]):

1. Replace the named binders, i.e., abstractions and let-constructions, with nameless binders by opening the bodies.
2. Cofinitely quantify the names introduced for variable opening.
3. Add premises to inductive rules in order to ensure that inductive judgements are restricted to locally closed terms.

We apply these steps to the inductive rules for the lazy natural semantics given in Figure 9. These rules produce judgements involving λ-terms as well as heaps. Hence, we also add premises that ensure that inductive judgements are restricted to well-formed heaps. The rules using the locally nameless representation are shown in Figure 10. For clarity, in the rules we put in braces the side-conditions to distinguish them better from the judgements.

The main difference with the rules in Figure 9 is the rule LNLet. To evaluate $\texttt{let}\ \overline{t}\ \texttt{in}\ t$ the local terms $\overline{t}$ have to be introduced in the heap, so that the body t is evaluated in this new context. To this purpose fresh names $\overline{x}$ are needed to open the local terms and the body. The evaluation of $t^{\overline{x}}$ produces a final heap and a value. Both are dependent on the names chosen for the local variables. The domain of the final heap consists of the local names $\overline{x}$ and the rest of names, say $\overline{z}$. The rule LNLet is cofinite quantified. As it is explained in [4], the advantage of the cofinite rules over existential and universal ones is that the freshness side-conditions are not explicit. The freshness condition for $\overline{x}$ is *hidden* in the finite set L, which includes the names that should be avoided during the reduction. The novelty of our cofinite rule, compared with the ones appearing in [1] and [4] (that are similar to the cofinite rules for the predicate `lc` in Figure 5), is that the names introduced in the (infinite) premises do appear in the conclusion too. Therefore, in the conclusion of the rule LNLet we can replace the names $\overline{x}$ by any list $\overline{y}$ not in L.

The problem with explicit freshness conditions is that they are associated just to rule instances, while they should apply to the whole reduction proof. Take for instance the rule LNVar. In the premise the binding $x \mapsto t$ does no longer belong to the heap. Therefore, a valid reduction for this premise may chose x as fresh. We avoid this situation by requiring that x is undefined in the final heap too.[6] By contrast to the rule Var in Figure 9, no renaming of the final value, that is w, is needed.

The side-condition of rule LNApp deserves an explanation too. Let us suppose that x is undefined in the initial heap Γ. We must avoid that x is chosen as a fresh name during the evaluation of t. For this reason we require that x is defined in the final heap Δ only if x was already defined in Γ. Notice how the body of the abstraction, that is u, is open with the name x. This is equivalent to the substitution of x for y in the body of the abstraction $\lambda y.e'$ (see rule App in Figure 9).

A *regularity* lemma ensures that the judgements produced by this reduction system involve only well-formed heaps and locally closed terms.

Lemma 5.
REGULARITY $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w.$

Similarly, Theorem 1 in [6] ensures that the property of being *distinctly named* is preserved by the rules in Figure 9.

The next lemma states that names defined in a context heap remain defined after the evaluation of any term in that context.

Lemma 6.
DEF_NOT_LOST $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Delta).$

[6] An alternative is to decorate judgements with a set collecting the names that have been taken out of the heap during a reduction proof, and starting with the empty set. This approach has been adopted by Sestoft in [7].

Moreover, fresh names are only introduced by the rule LNLet and, consequently, they are bound in the final heap/value pair. Therefore, any undefined free variable appearing in the final heap/value pair must occur in the initial heap/term pair too.

Lemma 7.
ADD_VARS $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow (x \in \mathtt{names}(\Delta : w) \Rightarrow (x \in \mathtt{dom}(\Delta) \vee x \in \mathtt{names}(\Gamma : t)))$.

A *renaming* lemma ensures that the evaluation of a term is independent of the fresh names chosen in the reduction process. Moreover, any name in the context can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reduction proofs for heap/term pairs are unique up to α-conversion of the names defined in the context heap.

Lemma 8.
RENAMING $\quad \Gamma : t \Downarrow \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w) \Rightarrow \Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$.

In addition, the renaming lemma permits to prove an *introduction* lemma for the cofinite rule LNLet which establishes that the corresponding existential rule is admissible too.

Lemma 9.
LET_INTRO $\quad (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \wedge \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ (\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t) \Rightarrow \Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}}$.

This result, together with the renaming lemma, justifies that our rule LNLet is equivalent to Launchbury's rule Let used with normalized terms.

4 Conclusions and future work

In the present work we have used a locally nameless representation not only for the pure λ-calculus [4] but also for its extension with mutually recursive local declarations. This notation avoids name clashing between bound and free variables. Afterwards, we have used this representation for redefining Launchbury's natural semantics for lazy evaluation [6]. To this purpose we have adapted the definition of context heaps to the locally nameless notation. A heap may be seen as a multiple binder. Actually, the names defined (bound) in a heap can be replaced by other names, provided that terms keep their meaning in the context represented by the heap. Our renaming lemma ensures that whenever a heap is renamed with fresh names, reduction proofs are preserved.

Launchbury assumes Barendregt's variable convention [2] in [6] when defining his operational semantics only for normalized λ-terms. In order to the avoid this problematic [8] variable convention, we have used cofinite quantification in our locally nameless reduction rules. Freshness conditions are usually considered in each rule individually. Nevertheless, this technique produces name clashing when

considering whole reduction proofs. A solution might be to decorate each rule with the set of forbidden names and indicate how to modify this set during the reduction process. However, this could be too restrictive in many occasions. Moreover, existential rules are not easy to deal with because each reduction is obtained just for one specific list of names. If any of the names in this list causes a name clashing with other reduction proofs, then it is cumbersome to demonstrate that an alternative reduction for a fresh list does exist. Cofinite quantification has allowed us to solve this problem because in a single step reductions are guaranteed for an infinite number of lists of names. Moreover, our introduction lemma guarantees that a more conventional exists-fresh rule is correct in our reduction system too.

The cofinite quantification that we have used in our semantic rules is more complex than those in [1] and [4]. Our cofinite rule LNLet in Figure 10 introduces quantified variables in the conclusion as well, as the latter depends on the chosen names.

Our future tasks include the implementation in the proof assistant Coq [3] of the natural semantics redefined in this paper. The final aim is to prove automatically the equivalence of the natural semantics with the alternative version given also in [6]. This alternative version differs from the original one in the introduction of indirections during β-reduction and the elimination of updates. At present we are working on the definition (using the locally nameless representation) of two intermediate semantics, one introducing indirections and the other without updates. Then, we will establish equivalence relations between heaps obtained by each semantics, which allow us to prove the equivalence of the original natural semantics and the alternative semantics through the intermediate semantics.

5 Acknowledgments

This work is partially supported by the projects: TIN2009-14599-C03-01 and S2009/TIC-1465.

References

1. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages, POPL'08*, pages 3–15. ACM Press, 2008.
2. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
3. Y. Bertot. Coq in a hurry. *CoRR*, abs/cs/0603118, 2006.
4. A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011.
5. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

6. J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.
7. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.
8. C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, pages 35–50. LNCS 4603, Springer-Verlag, 2007.

6 Appendix

6.1 Proof of Lemma 1: LC_AT_M_FROM_N

Lemma 1 :
LC_AT_M_FROM_N $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \Rightarrow \forall \overline{m} \geq \overline{n}\,.\,\texttt{lc\_at}\ k\ \overline{m}\ t$

Proof. The proof is done by structural induction on t.

- $t \equiv \texttt{bvar}\ i\ j$.
  $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)$, then $i < k\ \wedge\ j < \texttt{List.nth}\ i\ \overline{n}$.
  If $\overline{m} \geq \overline{n}$, then $\texttt{List.nth}\ i\ \overline{m} \geq \texttt{List.nth}\ i\ \overline{n}$.
  Consequently $i < k\ \wedge\ j < \texttt{List.nth}\ i\ \overline{m}$.
  Applying rule LCK-BVAR, $\texttt{lc\_at}\ k\ \overline{m}\ (\texttt{bvar}\ i\ j)$.

- $t \equiv \texttt{fvar}\ x$.
  Trivial.

- $t \equiv \texttt{abs}\ t'$.
  $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t')$, then $\texttt{lc\_at}\ (k+1)\ [1:\overline{n}]\ t'$.
  Since $\overline{m} \geq \overline{n}$, then $[1:\overline{m}] \geq [1:\overline{n}]$.
  By induction hypothesis, $\texttt{lc\_at}\ (k+1)\ [1:\overline{m}]\ t'$.
  Applying rule LCK-ABS, $\texttt{lc\_at}\ k\ \overline{m}\ (\texttt{abs}\ t)$.

- $t \equiv \texttt{app}\ t'\ v$.
  $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t'\ v)$, then $\texttt{lc\_at}\ k\ \overline{n}\ t' \wedge \texttt{lc\_at}\ k\ \overline{n}\ v$.
  Since $\overline{m} \geq \overline{n}$,
  by induction hypothesis, $\texttt{lc\_at}\ k\ \overline{m}\ t' \wedge \texttt{lc\_at}\ k\ \overline{m}\ v$.
  Applying rule LCK-APP, $\texttt{lc\_at}\ k\ \overline{m}\ (\texttt{app}\ t'\ v)$.

- $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
  $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$, then $\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ \overline{t} \wedge \texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ t'$.
  Since $\overline{m} \geq \overline{n}$, then $[|\overline{t}| : \overline{m}] \geq [|\overline{t}| : \overline{n}]$.
  By induction hypothesis, $\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{m}]\ \overline{t} \wedge \texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{m}]\ t'$.
  Applying rule LCK-LET, $\texttt{lc\_at}\ k\ \overline{m}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$.

□

6.2 Proof of Lemma 2: LC_IIF_LC_AT

To prove Lemma 2, we have to prove two auxiliary results: Lemmas 10 and 11.

If a term t opened with names $\overline{x}$ at level k is locally closed at level k with $\overline{n}$, then the term t is also locally closed at level $k+1$ with $[\overline{n} : |\overline{x}|]$.

Lemma 10.
LC_AT_K+1_FROM_K $\quad k = |\overline{n}| \wedge \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}t) \Rightarrow \texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ t$

Proof. The proof is done by induction on the structure of t.

- $t \equiv \texttt{bvar}\ i\ j$.
  $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{bvar}\ i\ j))$.
  - $i = k \wedge j < |\overline{x}|$
    By hypothesis, $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}))$
    Thus,
    $i = k < k+1 \wedge j < |\overline{x}| \overset{k=|\overline{n}|}{=} \texttt{List.nth}\ k\ [\overline{n} : |\overline{x}|] = \texttt{List.nth}\ i\ [\overline{n} : |\overline{x}|]$.
  - otherwise
    By hypothesis, $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)$, then $i < k \wedge j < \texttt{List.nth}\ i\ \overline{n}$.
    Thus, $i < k < k+1 \wedge j < \texttt{List.nth}\ i\ \overline{n} = \texttt{List.nth}\ i\ [\overline{n} : |\overline{x}|]$.

  In both cases, by rule LCK-BVAR, $\texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ (\texttt{bvar}\ i\ j)$.

- $t \equiv \texttt{fvar}\ x$.
  Trivial.

- $t \equiv \texttt{abs}\ t'$.
  Since $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{abs}\ t'))$, $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ (\{k+1 \to \overline{x}\}t'))$.
  Thus, $\texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ (\{k+1 \to \overline{x}\}t')$.
  By induction hypothesis, $\texttt{lc\_at}\ (k+2)\ [1 : \overline{n} : |\overline{x}|]\ t'$.
  Applying rule LCK-ABS, $\texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ (\texttt{abs}\ t')$.

- $t \equiv \texttt{app}\ t'\ v$.
  Since $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{app}\ t'\ v))$,
  $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ (\{k \to \overline{x}\}t')\ (\{k \to \overline{x}\}v))$.
  Thus, $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}t') \wedge \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}v)$.
  By induction hypothesis, $\texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ t' \wedge \texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ v$.
  Applying rule LCK-APP, $\texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ (\texttt{app}\ t'\ v)$.

- $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
  Since $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t'))$,
  $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ (\{k+1 \to \overline{x}\}\overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}t'))$.
  Thus, $\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ (\{k+1 \to \overline{x}\}\overline{t})$
  and $\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ (\{k+1 \to \overline{x}\}t')$.
  By induction hypothesis,
  $\texttt{lc\_at}\ (k+2)\ [|\overline{t}| : \overline{n} : |\overline{x}|]\ \overline{t} \wedge \texttt{lc\_at}\ (k+2)\ [|\overline{t}| : \overline{n} : |\overline{x}|]\ t'$.
  Applying rule LCK-LET, $\texttt{lc\_at}\ (k+1)\ [\overline{n} : |\overline{x}|]\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$.

□

Next lemma indicates that if a term is locally closed at level $k+1$ for a given list of natural numbers $[\overline{n} : n]$, then the term open with distinct fresh names $\overline{x}$ (such that $\overline{x} \geq n$) is closed at level k for the list $\overline{n}$.

Lemma 11.
LC_AT_K_FROM_K+1 $\quad k = |\overline{n}| \ \wedge \texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ t$
$\Rightarrow \forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}t)$

Proof. By structural induction on t.

– $t \equiv \texttt{bvar}\ i\ j$.
Since $\texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ (\texttt{bvar}\ i\ j)$, $i < k+1 \wedge j < \texttt{List.nth}\ i\ [\overline{n} : n]$
- $i = k \wedge j < \texttt{List.nth}\ k\ [\overline{n} : n] \overset{k=|\overline{n}|}{=} n$
  Let $\overline{x} \subseteq Id$ such that $|\overline{x}| \geq n$.
  Since $\{k \to \overline{x}\}(\texttt{bvar}\ k\ j) = \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x})$,
  applying rule LCK-FVAR, $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{bvar}\ i\ j))$.
- $i < k \wedge j < \texttt{List.nth}\ i\ [\overline{n} : n] = \texttt{List.nth}\ i\ \overline{n}$
  By rule LCK-BVAR, $\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{bvar}\ i\ j))$.

– $t \equiv \texttt{fvar}\ x$.
Trivial.

– $t \equiv \texttt{abs}\ t'$.
Since $\texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ (\texttt{abs}\ t')$, $\texttt{lc\_at}\ (k+2)\ [1 : \overline{n} : n]\ t'$.
By induction hypothesis,
$\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ (\{k+1 \to \overline{x}\}t')$.
Applying rule LCK-ABS,
$\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ (\{k+1 \to \overline{x}\}t'))$.
Thus, $\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{abs}\ t'))$.

– $t \equiv \texttt{app}\ t'\ v$.
Since $\texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ (\texttt{app}\ t'\ v)$,
$\texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ t'$ and $\texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ v$.
By induction hypothesis,
$\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, (\texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}t') \wedge \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}v))$.
Applying rule LCK-APP,
$\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ (\{k \to \overline{x}\}t')\ (\{k \to \overline{x}\}v))$.
Thus, $\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{app}\ t'\ v))$.

– $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
Since $\texttt{lc\_at}\ (k+1)\ [\overline{n} : n]\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$,
$\texttt{lc\_at}\ (k+2)\ [|\overline{t}| : \overline{n} : n]\ \overline{t}$ and $\texttt{lc\_at}\ (k+2)\ [|\overline{t}| : \overline{n} : n]\ t'$.
By induction hypothesis, $\forall \overline{x} \subseteq Id, |\overline{x}| \geq n$,
$\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ (\{k+1 \to \overline{x}\}\overline{t}) \wedge \texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ (\{k+1 \to \overline{x}\}t')$.
Applying rule LCK-LET,
$\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ (\{k+1 \to \overline{x}\}\overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}t'))$.
Thus, $\forall \overline{x} \subseteq Id, |\overline{x}| \geq n, \texttt{lc\_at}\ k\ \overline{n}\ (\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t'))$.

□

Now we are ready to prove that a term is locally closed if and only if is closed at level 0.

Lemma 2 :
LC_IIF_LC_AT $\quad \texttt{lc}\ t \Leftrightarrow \texttt{lc\_at}\ 0\ [\,]\ t$

Proof.
$\Rightarrow$) By structural induction on t:

- $t \equiv \texttt{bvar}\ i\ j$.
  Trivial.

- $t \equiv \texttt{fvar}\ x$.
  Trivial.

- $t \equiv \texttt{abs}\ t'$
  $\texttt{lc}\ (\texttt{abs}\ t')$, then $\forall x \notin L \subseteq Id.\texttt{lc}\ t'^{[x]}$.
  By induction hypothesis, $\forall x \notin L \subseteq Id.\texttt{lc\_at}\ 0\ [\,]\ t'^{[x]}$.
  Thus, $\forall x \notin L \subseteq Id.\texttt{lc\_at}\ 0\ [\,]\ (\{0 \to x\}t')$.
  By LC_AT_K+1_FROM_K, $\forall x \notin L \subseteq Id.\texttt{lc\_at}\ 1\ [1]\ t'$.
  By LCK-ABS, $\texttt{lc\_at}\ 0\ [\,]\ (\texttt{abs}\ t')$.

- $t \equiv \texttt{app}\ t'\ v$.
  $\texttt{lc}\ (\texttt{app}\ t'\ v)$, then $\texttt{lc}\ t'$ and $\texttt{lc}\ v$.
  By induction hypothesis, $\texttt{lc\_at}\ 0\ [\,]\ t'$ and $\texttt{lc\_at}\ 0\ [\,]\ v$.
  By LCK-APP, $\texttt{lc\_at}\ 0\ [\,]\ (\texttt{app}\ t'\ v)$.

- $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
  $\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$, then $\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id.\texttt{lc}\ [t : \overline{t}]^{\overline{x}}$.
  By induction hypothesis, $\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id.\texttt{lc\_at}\ 0\ [\,]\ [t : \overline{t}]^{\overline{x}}$.
  Thus, $\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id.\texttt{lc\_at}\ 0\ [\,]\ \{0 \to \overline{x}\}[t : \overline{t}]$.
  By LC_AT_K+1_FROM_K, $\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id.\texttt{lc\_at}\ 1\ [|\overline{x}|]\ [t : \overline{t}]$.
  By LCK-LET, $\texttt{lc\_at}\ 0\ [\,]\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$.

$\Leftarrow$) By structural induction on t:

- $t \equiv \texttt{bvar}\ i\ j$.
  Trivial, since this case is not possible.
  $\texttt{lc\_at}\ 0\ [\ ]\ (\texttt{bvar}\ i\ j) \Rightarrow i < 0 \wedge j < \texttt{List.nth}\ i\ [\ ]$, the empty list has no elements.

- $t \equiv \texttt{fvar}\ x$.
  Trivial.

– $t \equiv \texttt{abs}\ t'$.
Since $\texttt{lc\_at}\ 0\ [\,]\ (\texttt{abs}\ t')$, $\texttt{lc\_at}\ 1\ [1]\ t'$.
By LC_AT_K_FROM_K+1, $\forall \overline{x} \subseteq Id, |\overline{x}| \geq 1.\texttt{lc\_at}\ 0\ [\,]\ (\{0 \to \overline{x}\}t')$.
Thus, $\forall x \notin \emptyset \subseteq Id.\texttt{lc\_at}\ 0\ [\,]\ t'^{[x]}$.
By induction hypothesis, $\forall x \notin \emptyset \subseteq Id.\texttt{lc}\ t'^{[x]}$.
By LC-ABS, $\texttt{lc}\ (\texttt{abs}\ t')$.

– $t \equiv \texttt{app}\ t'\ v$.
Since $\texttt{lc\_at}\ 0\ [\,]\ (\texttt{app}\ t'\ v)$, $\texttt{lc\_at}\ 0\ [\,]\ t' \wedge \texttt{lc\_at}\ 0\ [\,]\ v$.
By induction hypothesis, $\texttt{lc}\ t' \wedge \texttt{lc}\ v$.
By LC-APP, $\texttt{lc}\ (\texttt{app}\ t'\ v)$.

– $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
Since $\texttt{lc\_at}\ 0\ [\,]\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$, $\texttt{lc\_at}\ 1\ [|\overline{t}|]\ \overline{t} \wedge \texttt{lc\_at}\ 1\ [|\overline{t}|]\ t'$.
By LC_AT_K_FROM_K+1, $\forall \overline{x} \subseteq Id, |\overline{x}| \geq |\overline{t}|$
$(\texttt{lc\_at}\ 0\ [\,]\ (\{0 \to \overline{x}\}\overline{t}) \wedge \texttt{lc\_at}\ 0\ [\,]\ (\{0 \to \overline{x}\}t'))$.
Thus, $\forall \overline{x}^{|\overline{t}|} \notin \emptyset \subseteq Id.(\texttt{lc\_at}\ 0\ [\,]\ \overline{t}^{\overline{x}} \wedge \texttt{lc\_at}\ 0\ [\,]\ t'^{\overline{x}})$.
By induction hypothesis, $\forall \overline{x}^{|\overline{t}|} \notin \emptyset \subseteq Id.(\texttt{lc}\ \overline{t}^{\overline{x}} \wedge \texttt{lc}\ t'^{\overline{x}})$.
By LC-LET, $\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$.

□

6.3 Proof of Lemma 3: CLOSE_OPEN_VAR and OPEN_CLOSE_VAR

Lemma 3 states that variable opening and variable closing are inverse functions under some side conditions. Its proof requires another two auxiliary lemmas. Lemma 12 expresses that opening a term at level k and then closing the result at the same level with the same names produces the original term whenever the chosen names to develop the opening and closing operations are fresh in the term.

Lemma 12.
CLOSE_OPEN_VAR_K $\quad \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow \{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}t) = t$

Proof. By structural induction on t:

– $t \equiv \texttt{bvar}\ i\ j$.

$$\begin{aligned}\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{bvar}\ i\ j)) &= \begin{cases} \{k \leftarrow \overline{x}\}(\texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x})) & \text{if } i = k \wedge j < |\overline{x}| \\ \{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & \text{otherwise} \end{cases} \\ &= \texttt{bvar}\ i\ j.\end{aligned}$$

– $t \equiv \texttt{fvar}\ x$.
If $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{fvar}\ x)$, then $x \notin \overline{x}$.
Thus, $\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{bvar}\ i\ j)) = \{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) = \texttt{fvar}\ x$.

– $t \equiv \texttt{abs}\ t'$.
If $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{abs}\ t')$, then $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t'$. Thus,

$$\begin{aligned}\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{abs}\ t')) &= \{k \leftarrow \overline{x}\}(\texttt{abs}\ (\{k+1 \to \overline{x}\}t')) \\ &= \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}(\{k+1 \to \overline{x}\}t')) \\ \text{I.H.} \qquad &= \texttt{abs}\ t'.\end{aligned}$$

– $t \equiv \texttt{app}\ t'\ v$.
If $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{app}\ t'\ v)$, then $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t'$ and $\texttt{fresh}\ \overline{x}\ \texttt{in}\ v$. Thus,

$$\begin{aligned}\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{app}\ t'\ v)) &= \{k \leftarrow \overline{x}\}(\texttt{app}\ (\{k \to \overline{x}\}t')\ (\{k \to \overline{x}\}v)) \\ &= \texttt{app}\ (\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}t'))\ (\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}v)) \\ \text{I.H.} \qquad &= \texttt{app}\ t'\ v.\end{aligned}$$

– $t \equiv \texttt{let}\ \bar{t}\ \texttt{in}\ t'$.
If $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{let}\ \bar{t}\ \texttt{in}\ t')$, then $\texttt{fresh}\ \overline{x}\ \texttt{in}\ \bar{t}$ and $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t'$. Thus,

$$\begin{aligned}&\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{let}\ \bar{t}\ \texttt{in}\ t')) \\ &\quad = \{k \leftarrow \overline{x}\}(\texttt{let}\ (\{k+1 \to \overline{x}\}\bar{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}t')) \\ &\quad = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}(\{k+1 \to \overline{x}\}\bar{t}))\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}(\{k+1 \to \overline{x}\}t')) \\ \text{I.H.} \quad &= \texttt{let}\ \bar{t}\ \texttt{in}\ t'.\end{aligned}$$

□

The second result (Lemma 13) establishes that closing a term at level k and then opening the result with the same names at the same level gives back the original term, when the term is closed at level k.

Lemma 13.
OPEN_CLOSE_VAR_K $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \Rightarrow \{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}t) = t$

Proof. By structural induction on t:

– $t \equiv \texttt{bvar}\ i\ j$.
If $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)$, then $i < k$ and $j < \texttt{List.nth}\ i\ \overline{n}$.
Thus, $\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j)) = \{k \to \overline{x}\}(\texttt{bvar}\ i\ j) = \texttt{bvar}\ i\ j$.

– $t \equiv \texttt{fvar}\ x$.

$$\begin{aligned}&\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x)) \\ &\qquad = \begin{cases}\{k \to \overline{x}\}(\texttt{bvar}\ k\ j) & \text{if } \exists j : 0 \le j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \{k \to \overline{x}\}(\texttt{fvar}\ x) & \text{otherwise}\end{cases} \\ &\qquad = \begin{cases}\texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } \exists j : 0 \le j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise}\end{cases} \\ &\qquad = \texttt{fvar}\ x.\end{aligned}$$

– $t \equiv \texttt{abs}\ t'$.
If $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t')$, then $\texttt{lc\_at}\ (k+1)\ [1:\overline{n}]\ t'$. Thus,

$$\begin{aligned}\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}(\texttt{abs}\ t')) &= \{k \to \overline{x}\}(\texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}t')) \\ &= \texttt{abs}\ (\{k+1 \to \overline{x}\}(\{k+1 \leftarrow \overline{x}\}t')) \\ \text{I.H.} \qquad &= \texttt{abs}\ t'\end{aligned}$$

– $t \equiv \texttt{app}\ t'\ v$.
If $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t'\ v)$, then $\texttt{lc\_at}\ k\ \overline{n}\ t'$ and $\texttt{lc\_at}\ k\ \overline{n}\ v$. Thus,

$$\begin{aligned}\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}(\texttt{app}\ t'\ v)) &= \{k \to \overline{x}\}(\texttt{app}\ (\{k \leftarrow \overline{x}\}t')\ (\{k \leftarrow \overline{x}\}v)) \\ &= \texttt{app}\ (\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}t'))\ (\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}v)) \\ \text{I.H.} \qquad &= \texttt{app}\ t'\ v\end{aligned}$$

– $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
If $\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t')$, then
$\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ \overline{t}$ and $\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ t'$. Thus,

$$\begin{aligned}&\{k \to \overline{x}\}(\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t')) \\ &\quad = \{k \to \overline{x}\}(\texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}t')) \\ &\quad = \texttt{let}\ (\{k+1 \to \overline{x}\}(\{k+1 \leftarrow \overline{x}\}\overline{t}))\ \texttt{in}\ (\{k+1 \to \overline{x}\}(\{k+1 \leftarrow \overline{x}\}t')) \\ \text{I.H.} \quad &= \texttt{let}\ \overline{t}\ \texttt{in}\ t'.\end{aligned}$$

□

Now the proof of Lemma 3 is straightforward.

Lemma 3

CLOSE_OPEN_VAR $\quad \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow {}^{\backslash\overline{x}}(t^{\overline{x}}) = t$

OPEN_CLOSE_VAR $\quad \texttt{lc}\ t \Rightarrow ({}^{\backslash\overline{x}}t)^{\overline{x}} = t$

Proof.

– CLOSE_OPEN_VAR is a corollary of Lemma 12 (take $k = 0$).
– OPEN_CLOSE_VAR is a corollary of Lemma 13 (take $k = 0$).

□

6.4 Proof of Lemma 4: OK_SUBS_OK

Lemmas 14 and 15 are needed to prove Lemma 4. Every variable in the domain of a heap where variable x has been substituted by y is either in the domain of the original heap, or coincides with y.

Lemma 14.

DOM_SUBS_UNION $\quad \texttt{dom}(\Gamma[y/x]) \subseteq \texttt{dom}(\Gamma) \cup \{y\}$

Proof. By induction on the size of Γ:

– $\Gamma = \emptyset$. Trivial.

– $\Gamma = (\Delta, z \mapsto t)$.
$\mathtt{dom}(\Gamma[y/x]) = \mathtt{dom}((\Delta[y/x], z[y/x] \mapsto t[y/x])) = \mathtt{dom}(\Delta[y/x]) \cup \{z[y/x]\}$
$\overset{IH}{\subseteq} \mathtt{dom}(\Delta) \cup \{y\} \cup \{z[y/x]\}$

- $z = x$.
$\mathtt{dom}(\Gamma[y/x]) \subseteq \mathtt{dom}(\Delta) \cup \{y\} \cup \{y\} \subseteq \mathtt{dom}(\Delta) \cup \{y\} \cup \{z\} = \mathtt{dom}(\Gamma) \cup \{y\}$

- $z \neq x$.
$\mathtt{dom}(\Gamma[y/x]) \subseteq \mathtt{dom}(\Delta) \cup \{y\} \cup \{z\} = \mathtt{dom}(\Gamma) \cup \{y\}$

□

Next lemma establishes that substitution preserves local closure

Lemma 15.
LC_SUBS_LC $\quad \mathtt{lc}\ t \Rightarrow \mathtt{lc}\ t[y/x]$

Proof. By structural induction on t:

– $t \equiv \mathtt{bvar}\ i\ j$.
Trivial.

– $t \equiv \mathtt{fvar}\ x$.
Trivial.

– $t \equiv \mathtt{abs}\ t'$.
$\mathtt{lc}\ (\mathtt{abs}\ t') \Rightarrow \forall z \notin L \subseteq Id\ .\ \mathtt{lc}\ t'^{[z]}$.
Let $L' = L \cup \{x\} \Rightarrow \forall z \notin L' \subseteq Id\ .\ \mathtt{lc}\ t'^{[z]}$.
By induction hypothesis, $\forall z \notin L' \subseteq Id\ .\ \mathtt{lc}\ (t'^{[z[y/x]]})$.
Since $z \neq x$, $\forall z \notin L' \subseteq Id\ .\ \mathtt{lc}\ (t'[y/x])^{[z]}$.
By LC-ABS, $\mathtt{lc}\ (\mathtt{abs}\ (t'[y/x]))$.
Thus, $\mathtt{lc}\ (\mathtt{abs}\ t')[y/x]$.

– $t \equiv \mathtt{app}\ t'\ v$.
$\mathtt{lc}\ (\mathtt{app}\ t'\ v) \Rightarrow \mathtt{lc}\ t' \wedge \mathtt{lc}\ v$.
By induction hypothesis, $\mathtt{lc}\ t'[y/x] \wedge \mathtt{lc}\ v[y/x]$.
By LC-APP, $\mathtt{lc}\ \mathtt{app}\ (t'[y/x])\ (v[y/x])$.
Thus, $\mathtt{lc}\ (\mathtt{app}\ t'\ v)[y/x]$.

– $t \equiv \mathtt{let}\ \overline{t}\ \mathtt{in}\ t'$.
$\mathtt{lc}\ \mathtt{let}\ \overline{t}\ \mathtt{in}\ t' \Rightarrow \forall \overline{z}^{|\overline{t}|} \notin L \subseteq Id\ .\ \mathtt{lc}\ [t : \overline{t}]^{\overline{z}}$.
Let $L' = L \cup \{x\} \Rightarrow \forall \overline{z}^{|\overline{t}|} \notin L' \subseteq Id\ .\ \mathtt{lc}\ [t : \overline{t}]^{\overline{z}}$.
By induction hypothesis, $\forall \overline{z}^{|\overline{t}|} \notin L' \subseteq Id\ .\ \mathtt{lc}\ ([t : \overline{t}]^{\overline{z}}[y/x])$.
Since $x \notin \overline{z}$, $\forall \overline{z}^{|\overline{t}|} \notin L' \subseteq Id\ .\ \mathtt{lc}\ ([t : \overline{t}][y/x]^{\overline{z}})$.
By LC-LET, $\mathtt{lc}\ (\mathtt{let}\ (\overline{t}[y/x])\ \mathtt{in}\ (t'[y/x]))$.
Thus, $\mathtt{lc}\ (\mathtt{let}\ \overline{t}\ \mathtt{in}\ t')[y/x]$.

□

Now we can prove Lemma 4:

Lemma 4

OK_SUBS_OK $\quad \texttt{ok}\ \Gamma \wedge y \notin \texttt{dom}(\Gamma) \Rightarrow \texttt{ok}\ \Gamma[y/x]$

Proof. By rule induction on the size of Γ:

- $\Gamma = \emptyset$. Trivial.

- $\Gamma = (\Delta, z \mapsto t)$.
  $\texttt{ok}\ (\Delta, z \mapsto t) \Rightarrow \texttt{ok}\ \Delta \wedge z \notin \texttt{dom}(\Delta) \wedge \texttt{lc}\ t$.
  Let $y \notin \texttt{dom}(\Delta, z \mapsto t) = \texttt{dom}(\Delta) \cup \{z\} \Rightarrow y \notin \texttt{dom}(\Delta) \wedge y \neq z$.
  By induction hypothesis, $\texttt{ok}\ \Delta[y/x]$.
  - Case $z \neq x$:
    $\texttt{dom}(\Delta[y/x]) \overset{L14}{\subseteq} \texttt{dom}(\Delta) \cup \{y\}$.
    Since $z \notin \texttt{dom}(\Delta)$ and $z \neq y$, then $z \notin \texttt{dom}(\Delta[y/x])$.

  - Case $z = x$:
    $z = x \Rightarrow x \notin \texttt{dom}(\Delta) \Rightarrow \texttt{dom}(\Delta[y/x]) = \texttt{dom}(\Delta)$.
    Thus, $y \notin \texttt{dom}(\Delta[y/x])$.

  By Lemma 15, $\texttt{lc}\ t[y/x]$.
  Thus, $\texttt{ok}\ (\Delta, z \mapsto t)[y/x]$, i.e., $\texttt{ok}\ \Gamma[y/x]$.

□

6.5 Proof of Lemma 5: REGULARITY

Lemma 5

REGULARITY $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w$.

Proof. By rule induction:

- LNLam.
  Trivial.

- LNVar.
  By induction hypothesis, $\texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w$.
  Since $x \notin \texttt{dom}(\Gamma)$, $x \notin \texttt{dom}(\Delta)$,
  then $\texttt{ok}\ (\Gamma, x \mapsto t)$ and $\texttt{ok}\ (\Delta, x \mapsto w)$ and $\texttt{lc}\ (\texttt{fvar}\ x)$ by definition.

- LNApp.
  By induction hypothesis, $\texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Theta \wedge \texttt{lc}\ (\texttt{abs}\ u)$.
  By induction hypothesis, $\texttt{ok}\ \Theta \wedge \texttt{lc}\ u^{[x]} \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w$.
  Since $\texttt{lc}\ t$ and $\texttt{lc}\ (\texttt{fvar}\ x)$, then $\texttt{lc}\ (\texttt{app}\ t\ (\texttt{fvar}\ x))$.

– LNLet.
By induction hypothesis,
$\forall \overline{x}^{|\overline{t}|} \notin L.\mathtt{ok}\ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) \wedge \mathtt{lc}\ t^{\overline{x}} \wedge \mathtt{ok}\ (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) \wedge \mathtt{lc}\ w^{\overline{x}}$.
Particularly for $\overline{y}^{|\overline{t}|} \notin L.\mathtt{ok}\ (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) \wedge \mathtt{lc}\ w^{\overline{y}}$.
Since $\forall \overline{x}^{|\overline{t}|} \notin L.\mathtt{ok}\ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}})$, then $\mathtt{ok}\ \Gamma \wedge \forall \overline{x}^{|\overline{t}|} \notin L.(\overline{x} \notin \mathtt{dom}(\Gamma) \wedge \mathtt{lc}\ \overline{t}^{\overline{x}})$.
Since $\forall \overline{x}^{|\overline{t}|} \notin L.(\mathtt{lc}\ \overline{t}^{\overline{x}} \wedge \mathtt{lc}\ t^{\overline{x}})$, then $\mathtt{lc}\ (\mathtt{let}\ \overline{t}\ \mathtt{in}\ t)$.

□

6.6 Proofs of Lemmas 6 and 7: DEF_NOT_LOST and ADD_VARS

Lemma 6
DEF_NOT_LOST $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta)$.

Proof. By rule induction:

– LNLam.
Trivial.

– LNVar.
By induction hypothesis,
$\mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta) \Rightarrow \mathtt{dom}(\Gamma, x \mapsto t) \subseteq \mathtt{dom}(\Delta, x \mapsto w)$.

– LNApp.
By induction hypothesis, $\mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Theta)$ and $\mathtt{dom}(\Theta) \subseteq \mathtt{dom}(\Delta)$.
By transitivity, $\mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta)$.

– LNLet.
By induction hypothesis,
$\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ .\ \mathtt{dom}(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) \subseteq \mathtt{dom}(\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}})$.
Particularly for $\overline{y}^{|\overline{t}|} \notin L \subseteq Id$,
$\mathtt{dom}(\Gamma, \overline{y} \mapsto \overline{t}^{\overline{y}}) = \mathtt{dom}(\Gamma) \cup \{\overline{y}\} \subseteq \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}})$.
Thus, $\mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}})$.

□

Lemma 7
ADD_VARS $\quad \Gamma : t \Downarrow \Delta : w$
$\Rightarrow (x \in \mathtt{names}(\Delta : w) \Rightarrow (x \in \mathtt{dom}(\Delta) \vee x \in \mathtt{names}(\Gamma : t)))$.

Proof. It is equivalent to prove

$$\Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{names}(\Delta : w) \subseteq \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma : t).$$

By rule induction:

– LNLam.
Trivial.

– LNVar.
$\mathtt{names}((\Delta, x \mapsto w) : w) = \mathtt{names}(\Delta : w) \cup \{x\}$
$\overset{IH}{\subseteq} \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma : t) \cup \{x\}$
$= \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma) \cup \mathtt{fv}(t) \cup \{x\}$
$= \mathtt{dom}(\Delta, x \mapsto w) \cup \mathtt{names}(\Gamma, x \mapsto t) \cup \mathtt{fv}(\mathtt{fvar}\ x)$
$= \mathtt{dom}(\Delta, x \mapsto w) \cup \mathtt{names}((\Gamma, x \mapsto t) : \mathtt{fvar}\ x).$

– LNApp.
$\mathtt{names}(\Delta : w) \overset{IH}{\subseteq} \mathtt{dom}(\Delta) \cup \mathtt{names}(\Theta : u^{[x]})$
$\subseteq \mathtt{dom}(\Delta) \cup \mathtt{names}(\Theta) \cup \mathtt{fv}(u) \cup \{x\}$
$= \mathtt{dom}(\Delta) \cup \mathtt{names}(\Theta) \cup \mathtt{fv}(\mathtt{abs}\ u) \cup \mathtt{fv}(\mathtt{fvar}\ x)$
$= \mathtt{dom}(\Delta) \cup \mathtt{names}(\Theta : \mathtt{abs}\ u) \cup \mathtt{fv}(\mathtt{fvar}\ x)$
$\overset{IH}{\subseteq} \mathtt{dom}(\Delta) \cup \mathtt{dom}(\Theta) \cup \mathtt{names}(\Gamma : t) \cup \mathtt{fv}(\mathtt{fvar}\ x)$
$\overset{L6}{=} \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma : t) \cup \mathtt{fv}(\mathtt{fvar}\ x)$
$= \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma) \cup \mathtt{fv}(t) \cup \mathtt{fv}(\mathtt{fvar}\ x)$
$= \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma) \cup \mathtt{fv}(\mathtt{app}\ t\ (\mathtt{fvar}\ x))$
$= \mathtt{dom}(\Delta) \cup \mathtt{names}(\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x)).$

– LNLet.
$\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id$
$\mathtt{names}((\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^x}) : w^{\overline{x}}) \overset{IH}{\subseteq} \mathtt{dom}(\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^x}) \cup \mathtt{names}((\Gamma, \overline{x} \mapsto \overline{t^x}) : t^{\overline{x}}).$
Particularly for $\overline{y}^{|\overline{t}|} \notin L \subseteq Id$:
$\mathtt{names}((\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^y}) : w^{\overline{y}}) \overset{IH}{\subseteq} \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^y}) \cup \mathtt{names}((\Gamma, \overline{y} \mapsto \overline{t^y}) : t^{\overline{y}})$
$= \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^y}) \cup \mathtt{names}(\Gamma) \cup \{\overline{y}\} \cup \mathtt{fv}(\overline{t^y}) \cup \mathtt{fv}(t^{\overline{y}})$
$\subseteq \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^y}) \cup \mathtt{names}(\Gamma) \cup \{\overline{y}\} \cup \mathtt{fv}(\overline{t}) \cup \{\overline{y}\} \cup \mathtt{fv}(t) \cup \{\overline{y}\}$
$= \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^y}) \cup \mathtt{names}(\Gamma) \cup \mathtt{fv}(\mathtt{let}\ \overline{t}\ \mathtt{in}\ t)$
$= \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^y}) \cup \mathtt{names}(\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t).$

□

6.7 Proof of Lemma 8: RENAMING

Before proving the renaming lemma (Lemma 8) we need some auxiliary results: Corollaries 1 and 2, that are proved by Lemmas 16 and 17 respectively.

Lemma 16.
NOT_OPENK_FV $\quad \mathtt{fresh}\ y\ \mathtt{in}\ \{k \to \overline{x}\}t \Rightarrow \mathtt{fresh}\ y\ \mathtt{in}\ t$

Proof. By structural induction on t:

– $t \equiv \mathtt{bvar}\ i\ j$.
Trivial, since $\mathtt{fv}(\mathtt{bvar}\ i\ j) = \emptyset$.

– $t \equiv \mathtt{fvar}\ z$.
Trivial, since $\mathtt{fv}(\{k \to \overline{x}\}\mathtt{fvar}\ z) = \mathtt{fv}(\mathtt{fvar}\ z) = \{z\}$.

– $t \equiv \texttt{abs}\ t'$.
Since $\texttt{fresh}\ y\ \texttt{in}\ \{k \to \overline{x}\}(\texttt{abs}\ t')$,
$y \notin \texttt{fv}(\{k \to \overline{x}\}\texttt{abs}\ t') = \texttt{fv}(\texttt{abs}\ (\{k+1 \to \overline{x}\}t')) = \texttt{fv}(\{k+1 \to \overline{x}\}t')$.
By induction hypothesis, $y \notin \texttt{fv}(t') = \texttt{fv}(\texttt{abs}\ t')$.

– $t \equiv \texttt{app}\ t'\ v$.
Since $\texttt{fresh}\ y\ \texttt{in}\ \{k \to \overline{x}\}(\texttt{app}\ t'\ v)$,
$y \notin \texttt{fv}(\{k \to \overline{x}\}\texttt{app}\ t'\ v)$
$= \texttt{fv}(\texttt{app}\ (\{k \to \overline{x}\}t')\ (\{k \to \overline{x}\}v))$
$= \texttt{fv}(\{k \to \overline{x}\}t') \cup \texttt{fv}(\{k \to \overline{x}\}v)$.
By induction hypothesis, $y \notin \texttt{fv}(t') \wedge y \notin \texttt{fv}(v)$.
Therefore, $y \notin \texttt{fv}(t') \cup \texttt{fv}(v) = \texttt{fv}(\texttt{app}\ t'\ v)$.

– $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
Since $\texttt{fresh}\ y\ \texttt{in}\ \{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t')$,
$y \notin \texttt{fv}(\{k \to \overline{x}\}\texttt{let}\ \overline{t}\ \texttt{in}\ t')$
$= \texttt{fv}(\texttt{let}\ (\{k+1 \to \overline{x}\}\overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}t'))$
$= \texttt{fv}(\{k+1 \to \overline{x}\}\overline{t}) \cup \texttt{fv}(\{k+1 \to \overline{x}\}t')$.
By induction hypothesis, $y \notin \texttt{fv}(\overline{t}) \wedge y \notin \texttt{fv}(t')$.
Therefore, $y \notin \texttt{fv}(\overline{t}) \cup \texttt{fv}(t') = \texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t')$.

□

Corollary 1.
NOT_OPEN_FV $\quad \texttt{fresh}\ y\ \texttt{in}\ t^{\overline{x}} \Rightarrow \texttt{fresh}\ y\ \texttt{in}\ t$

Proof. This is a particular case of Lemma 16 ($k = 0$). □

Lemma 17.
FREE_VAR_OPENK $\quad \texttt{fresh}\ \overline{y}\ \texttt{in}\ t \wedge \overline{y} \cap \overline{x} = \emptyset \Rightarrow \texttt{fresh}\ \overline{y}\ \texttt{in}\ \{k \to \overline{x}\}t$

Proof. By structural induction on t:

– $t \equiv \texttt{bvar}\ i\ j$.
$\overline{y} \notin \texttt{fv}(t) \wedge \overline{y} \cap \overline{x} = \emptyset$.

$$\texttt{fv}(\{k \to \overline{x}\}(\texttt{bvar}\ i\ j)) = \begin{cases} \texttt{fv}(\texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x})) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{fv}(\texttt{bvar}\ i\ j) & \text{otherwise} \end{cases}$$
$$= \begin{cases} \texttt{List.nth}\ j\ \overline{x} & \text{if } i = k \wedge j < |\overline{x}| \\ \emptyset & \text{otherwise} \end{cases}$$

In both cases $\overline{y} \notin \texttt{fv}(\{k \to \overline{y}\}(\texttt{bvar}\ i\ j))$.

– $t \equiv \texttt{fvar}\ z$.
Trivial, since $\texttt{fv}(\{k \to \overline{x}\}\texttt{fvar}\ z) = \texttt{fv}(\texttt{fvar}\ z) = \{z\}$.

– $t \equiv \texttt{abs}\ t'$.
$\overline{y} \notin \texttt{fv}(\texttt{abs}\ t') = \texttt{fv}(t') \wedge \overline{y} \cap \overline{x} = \emptyset$.
By induction hypothesis, $\overline{y} \notin \texttt{fv}(\{k+1 \to \overline{x}\}t') = \texttt{fv}(\{k \to \overline{x}\}\texttt{abs}\ t')$.

– $t \equiv \texttt{app}\ t'\ v$.
$\overline{y} \notin \texttt{fv}(\texttt{app}\ t'\ v) = \texttt{fv}(t') \cup \texttt{fv}(v) \wedge \overline{y} \cap \overline{x}$.
By induction hypothesis $\overline{y} \notin \texttt{fv}(\{k \to \overline{x}\}t') \wedge \overline{y} \notin \texttt{fv}(\{k \to \overline{x}\}v)$.
Thus,
$y \notin \texttt{fv}(\{k \to \overline{x}\}t') \cup \texttt{fv}(\{k \to \overline{x}\}v)$
$= \texttt{fv}(\texttt{app}\ (\{k \to \overline{x}\}t')\ (\{k \to \overline{x}\}v))$
$= \texttt{fv}(\{k \to \overline{x}\}\texttt{app}\ t'\ v)$

– $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$.
$\overline{y} \notin \texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t') = \texttt{fv}(\overline{t}) \cup \texttt{fv}(t') \wedge \overline{y} \cap \overline{x}$.
By induction hypothesis $\overline{y} \notin \texttt{fv}(\{k+1 \to \overline{x}\}\overline{t}) \wedge \overline{y} \notin \texttt{fv}(\{k+1 \to \overline{x}\}t')$.
Thus,
$y \notin \texttt{fv}(\{k+1 \to \overline{x}\}\overline{t}) \cup \texttt{fv}(\{k+1 \to \overline{x}\}t')$
$= \texttt{fv}(\texttt{let}\ (\{k+1 \to \overline{x}\}\overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}t'))$
$= \texttt{fv}(\{k \to \overline{x}\}\texttt{let}\ \overline{t}\ \texttt{in}\ t')$.

□

Corollary 2.
FREE_VAR_OPEN $\quad \texttt{fresh}\ \overline{y}\ \texttt{in}\ t \wedge \overline{y} \cap \overline{x} = \emptyset \Rightarrow \texttt{fresh}\ \overline{y}\ \texttt{in}\ t^{\overline{x}}$

Proof. Take $k = 0$ in Lemma 17. □

Another auxiliary result is needed:

Lemma 18.
NOT_SUBS_DOM $\quad z \notin \texttt{dom}(\Gamma[y/x]) \wedge z \neq x \Rightarrow z \notin \texttt{dom}(\Gamma)$

Proof. By induction on the size of Γ:

– $\Gamma = \emptyset$. Trivial.

– $\Gamma = (\Delta, x' \mapsto t)$.
$\texttt{dom}(\Gamma[y/x]) = \texttt{dom}((\Delta[y/x], x'[y/x] \mapsto t[y/x])) = \texttt{dom}(\Delta[y/x]) \cup \{x'[y/x]\}$.
$z \notin \texttt{dom}(\Gamma[y/x]) = \texttt{dom}(\Delta[y/x]) \cup \{x'[y/x]\} \overset{IH}{\Rightarrow} z \notin \texttt{dom}(\Delta) \cup \{x'[y/x]\}$.
- $x' = x$.
$z \notin \texttt{dom}(\Delta) \cup \{y\} \overset{z \neq x}{\Rightarrow} z \notin \texttt{dom}(\Delta) \cup \{y\} \cup \{x\} = \texttt{dom}(\Gamma) \cup \{y\}$
$\Rightarrow z \notin \texttt{dom}(\Gamma)$.
- $x' \neq x$.
$z \notin \texttt{dom}(\Delta) \cup \{x'\} = \texttt{dom}(\Gamma)$.

□

The last auxiliary result that is needed establishes that if a variable x does not belong to the domain of a heap then the domain of the heap where x is substituted by y coincides with the domain of the heap:

Lemma 19.
DOM_SUBS $\quad x \notin \texttt{dom}(\Gamma) \Rightarrow \texttt{dom}(\Gamma[y/x]) = \texttt{dom}(\Gamma)$

Proof. By induction on the size of Γ:

- $\Gamma = \emptyset$. Trivial.

- $\Gamma = (\Delta, z \mapsto t)$.
  $x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin \mathtt{dom}(\Delta) \cup \{z\} \Rightarrow \begin{cases} x \notin \mathtt{dom}(\Delta) \overset{IH}{\Rightarrow} \mathtt{dom}(\Delta[y/x]) = \mathtt{dom}(\Delta) \\ x \neq z \end{cases}$
  $\mathtt{dom}(\Gamma[y/x]) = \mathtt{dom}(\Delta[y/x], z \mapsto t[y/x]) = \mathtt{dom}(\Delta[y/x]) \cup \{z\}$
  $= \mathtt{dom}(\Delta) \cup \{z\} = \mathtt{dom}(\Gamma)$

□

And now we prove the renaming lemma.

Lemma 8
RENAMING $\quad \Gamma : t \Downarrow \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w)$
$\Rightarrow \Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$.

Proof. By rule induction:

- LNLam.
  $\Gamma : \mathtt{abs}\ t \Downarrow \Gamma : \mathtt{abs}\ t \Rightarrow \{\mathtt{ok}\ \Gamma\} \wedge \{\mathtt{lc}\ \mathtt{abs}\ t\}$.
  $\mathtt{ok}\ \Gamma \wedge y \notin \mathtt{names}(\Gamma : \mathtt{abs}\ t) \Rightarrow \mathtt{ok}\ \Gamma \wedge y \notin \mathtt{dom}(\Gamma) \overset{L4}{\Rightarrow} \mathtt{ok}\ \Gamma[y/x]$.
  $\mathtt{lc}\ (\mathtt{abs}\ t) \overset{L15}{\Rightarrow} \mathtt{lc}\ (\mathtt{abs}\ t)[y/x]$.
  By rule LNLam, $\Gamma[y/x] : (\mathtt{abs}\ t)[y/x] \Downarrow \Gamma[y/x] : (\mathtt{abs}\ t)[y/x]$.

- LNVar.
  $(\Gamma, z \mapsto t) : (\mathtt{fvar}\ z) \Downarrow (\Delta, z \mapsto w) : w \Rightarrow$
  $\Gamma : t \Downarrow \Delta : w \wedge \{z \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta)\}$.
  $y \notin \mathtt{names}((\Gamma, z \mapsto t) : \mathtt{fv}(z)) \cup \mathtt{names}((\Delta, z \mapsto w) : w)$
  $= \mathtt{names}(\Gamma) \cup \mathtt{names}(\Delta) \cup \{z\} \cup \mathtt{fv}(t) \cup \mathtt{fv}(w)$
  $\Rightarrow y \notin \mathtt{names}(\Gamma) \cup \mathtt{names}(\Delta) \cup \mathtt{fv}(t) \cup \mathtt{fv}(w)$
  $\Rightarrow y \notin \mathtt{names}(\Gamma : t) \cup \mathtt{names}(\Delta : w)$.
  By induction hypothesis, $\Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$.
  To prove: $z[y/x] \notin \mathtt{dom}(\Gamma[y/x]) \cup \mathtt{dom}(\Delta[y/x])$

  1. $z \neq x \Rightarrow z \neq y$
     $\mathtt{dom}(\Gamma[y/x]) \cup \mathtt{dom}(\Delta[y/x]) \overset{L14}{\subseteq} \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta) \cup \{y\}$.
     $z \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta) \wedge y \neq z \Rightarrow z \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta) \cup \{y\}$
     $\Rightarrow z \notin \mathtt{dom}(\Gamma[y/x]) \cup \mathtt{dom}(\Delta[y/x])$.

  2. $z = x \Rightarrow z[y/x] = y$.
     $y \notin \mathtt{names}(\Gamma) \cup \mathtt{names}(\Delta)$
     $\Rightarrow y \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta) \overset{L19}{=} \mathtt{dom}(\Gamma[y/x]) \cup \mathtt{dom}(\Delta[y/x])$.

  By rule LNVar, $(\Gamma, z \mapsto t)[y/x] : (\mathtt{fvar}\ z)[y/x] \Downarrow (\Delta, z \mapsto w)[y/x] : w[y/x]$.

– LNApp.
$\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z) \Downarrow \Delta : w$
$\Rightarrow \Gamma : t \Downarrow \Theta : \texttt{abs}\ u \wedge \Theta : u^{[z]} \Downarrow \Delta : w \wedge \{z \notin \texttt{dom}(\Gamma) \Rightarrow z \notin \texttt{dom}(\Delta)\}$.

$$\begin{aligned}\texttt{names}(\Gamma : t) &\subseteq \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z))\\ &\subseteq \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z)) \cup \texttt{names}(\Delta : w).\end{aligned}$$

$$\begin{aligned}\texttt{names}(\Theta : \texttt{abs}\ u) &\overset{L7}{\subseteq} \texttt{dom}(\Theta) \cup \texttt{names}(\Gamma : t) \overset{L6}{\subseteq} \texttt{dom}(\Delta) \cup \texttt{names}(\Gamma : t)\\ &\subseteq \texttt{names}(\Delta) \cup \texttt{names}(\Gamma) \cup \texttt{fv}(t)\\ &\subseteq \texttt{names}(\Delta) \cup \texttt{names}(\Gamma) \cup \texttt{fv}(\texttt{app}\ t\ (\texttt{fvar}\ z)) \cup \texttt{fv}(w)\\ &= \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z)) \cup \texttt{names}(\Delta : w).\end{aligned}$$

$y \notin \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z)) \cup \texttt{names}(\Delta : w)$
$\Rightarrow y \notin \texttt{names}(\Gamma : t) \cup \texttt{names}(\Theta : \texttt{abs}\ u)$.

By induction hypothesis,

$$\Gamma[y/x] : t[y/x] \Downarrow \Theta[y/x] : \underbrace{(\texttt{abs}\ u)[y/x]}_{\texttt{abs}\ u[y/x]} \tag{1}$$

By OPEN_VAR_FV in [4] ($\texttt{fv}(u^{[z]}) \subseteq \texttt{fv}(u) \cup \{z\}$),
$\texttt{names}(\Theta : u^{[z]}) = \texttt{names}(\Theta) \cup \texttt{fv}(u^{[z]}) \subseteq \texttt{names}(\Theta) \cup \texttt{fv}(u) \cup \{z\}$.

$$\left.\begin{array}{l} y \notin \texttt{names}(\Theta : \texttt{abs}\ u) = \texttt{names}(\Theta) \cup \texttt{fv}(u)\\ y \notin \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z)) \Rightarrow y \neq z \end{array}\right\} \Rightarrow y \notin \texttt{names}(\Theta : u^{[z]}).$$

By induction hypothesis,

$$\Theta[y/x] : \underbrace{(u^{[z]})[y/x]}_{u[y/x]^{[z[y/x]]}} \Downarrow \Delta[y/x] : w[y/x] \tag{2}$$

To prove: $z[y/x] \notin \texttt{dom}(\Gamma[y/x]) \Rightarrow z[y/x] \notin \texttt{dom}(\Delta[y/x])$.

- $z \neq x \Rightarrow z[y/x] = z$
  $\texttt{dom}(\Delta[y/x]) \overset{L14}{\subseteq} \texttt{dom}(\Delta) \cup \{y\}$.
  $$\left.\begin{array}{l} z \notin \texttt{dom}(\Gamma[y/x]) \overset{L18}{\Rightarrow} z \notin \texttt{dom}(\Gamma) \overset{hip.}{\Rightarrow} z \notin \texttt{dom}(\Delta).\\ y \notin \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ z)) \Rightarrow y \neq z \end{array}\right\}$$
  $\Rightarrow z \notin \texttt{dom}(\Delta) \cup \{y\} \Rightarrow z \notin \texttt{dom}(\Delta[y/x])$
- $z = x \Rightarrow z[y/x] = y$.
  $y \notin \texttt{dom}(\Gamma[y/x]) \Rightarrow x \notin \texttt{dom}(\Gamma) \overset{hip.}{\Rightarrow} x \notin \texttt{dom}(\Delta) \overset{L19}{\Rightarrow} \texttt{dom}(\Delta) = \texttt{dom}(\Delta[y/x])$.
  $y \notin \texttt{names}(\Delta : w) \Rightarrow y \notin \texttt{dom}(\Delta) \Rightarrow y \notin \texttt{dom}(\Delta[y/x])$

Therefore,

$$z[y/x] \notin \texttt{dom}(\Gamma[y/x]) \Rightarrow z[y/x] \notin \texttt{dom}(\Delta[y/x]) \tag{3}$$

By 1, 2, 3 and rule LNApp, $\Gamma[y/x] : (\texttt{app}\ t\ (\texttt{fvar}\ z))[y/x] \Downarrow \Delta[y/x] : w[y/x]$.

– LNLET.
$\Gamma : \texttt{let}\ \bar{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}$
$\Rightarrow \forall \overline{x}^{|\bar{t}|} \notin L \subseteq Id.(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \wedge \{\overline{y}^{|\bar{t}|} \notin L \subseteq Id\}$.
CASE: $y \in L$.

- SUBCASE: $x \notin L$.
Let $L' = L \cup \{x\} - \{y\}$.
To prove: $\forall \overline{x} \notin L'$.
$(\Gamma[y/x], \overline{x} \mapsto \bar{t}[y/x]^{\overline{x}}) : t[y/x]^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z}[y/x] \mapsto \overline{u}[y/x]^{\overline{x}}) : w[y/x]^{\overline{x}}$
Let $\overline{x} \notin L'$.

SUBSUBCASE: $\overline{x} \cap \{y\} = \emptyset \Rightarrow \overline{x} \notin L \cup \{x\} \Rightarrow \overline{x} \cap \{x\} = \emptyset$.
$\overline{x} \notin L \cup \{x\} \Rightarrow (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}}$

$\overline{x} \cap \{y\} = \emptyset$
$\wedge y \notin \texttt{names}(\Gamma : \texttt{let}\ \bar{t}\ \texttt{in}\ t) \cup \texttt{names}((\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}})$
$= \texttt{names}(\Gamma) \cup \texttt{fv}(\bar{t}) \cup \texttt{fv}(t) \cup \overline{y} \cup \overline{z} \cup \texttt{fv}(\overline{u}^{\overline{y}}) \cup \texttt{fv}(w^{\overline{y}})$
$\overset{C1}{\Rightarrow} y \notin \texttt{names}(\Gamma) \cup \texttt{fv}(\bar{t}) \cup \texttt{fv}(t) \cup \overline{y} \cup \overline{z} \cup \texttt{fv}(\overline{u}) \cup \texttt{fv}(w)$
$\overset{C2}{\Rightarrow} y \notin \texttt{names}(\Gamma) \cup \texttt{fv}(\overline{t}^{\overline{x}}) \cup \texttt{fv}(t^{\overline{x}}) \cup \overline{y} \cup \overline{z} \cup \texttt{fv}(\overline{u}^{\overline{x}}) \cup \texttt{fv}(w^{\overline{x}}) \cup \overline{x}$
$\Rightarrow y \notin \texttt{names}((\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}}) \cup \texttt{names}((\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}})$.

By induction hypothesis,
$(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}})[y/x] : (t^{\overline{x}})[y/x] \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}})[y/x] : (w^{\overline{x}})[y/x] \overset{x \cap \overline{x} = \emptyset}{\Rightarrow}$
$(\Gamma[y/x], \overline{x} \mapsto \bar{t}[y/x]^{\overline{x}}) : t[y/x]^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z}[y/x] \mapsto \overline{u}[y/x]^{\overline{x}}) : w[y/x]^{\overline{x}}$.

SUBSUBCASE: $\overline{x} \cap \{y\} \neq \emptyset$.
Without lost of generality, consider $\overline{x} = [y : \overline{x'}]$ with $\overline{x'} \cap \{y\} = \emptyset$.
$\overline{x} \notin L' \Rightarrow \overline{x} \cap \{x\} = \emptyset$.
Let $\overline{x''} = [x : \overline{x'}] \Rightarrow \overline{x''} \notin L \Rightarrow$
$(\Gamma, [x : \overline{x'}] \mapsto \bar{t}^{[x:\overline{x'}]}) : t^{[x:\overline{x'}]} \Downarrow ([x : \overline{x'}] +\!\!+ \overline{z} \mapsto \overline{u}^{[x:\overline{x'}]}) : w^{[x:\overline{x'}]}$
$y \cap \overline{x''} = \emptyset$
$\wedge y \notin \texttt{names}(\Gamma : \texttt{let}\ \bar{t}\ \texttt{in}\ t) \cup \texttt{names}((\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}})$
$= \texttt{names}(\Gamma) \cup \texttt{fv}(\bar{t}) \cup \texttt{fv}(t) \cup \overline{y} \cup \overline{z} \cup \texttt{fv}(\overline{u}^{\overline{y}}) \cup \texttt{fv}(w^{\overline{y}})$
$\overset{C1}{\Rightarrow} y \notin \texttt{names}(\Gamma) \cup \texttt{fv}(\bar{t}) \cup \texttt{fv}(t) \cup \overline{y} \cup \overline{z} \cup \texttt{fv}(\overline{u}) \cup \texttt{fv}(w)$
$\overset{C2}{\Rightarrow} y \notin \texttt{names}(\Gamma) \cup \texttt{fv}(\overline{t}^{\overline{x''}}) \cup \texttt{fv}(t^{\overline{x''}}) \cup \overline{y} \cup \overline{z} \cup \texttt{fv}(\overline{u}^{\overline{x''}}) \cup \texttt{fv}(w^{\overline{x''}}) \cup \overline{x''}$
$\Rightarrow y \notin \texttt{names}((\Gamma, \overline{x''} \mapsto \overline{t}^{\overline{x''}}) : t^{\overline{x''}}) \cup \texttt{names}((\overline{x''} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x''}}) : w^{\overline{x''}})$.

By induction hypothesis,
$(\Gamma, [x : \overline{x'}] \mapsto \bar{t}^{[x:\overline{x'}]})[y/x] : (t^{[x:\overline{x'}]})[y/x] \Downarrow ([x : \overline{x'}] +\!\!+ \overline{z} \mapsto \overline{u}^{[x:\overline{x'}]})[y/x] :$
$(w^{[x:\overline{x'}]})[y/x] \Rightarrow$
$(\Gamma[y/x], [y : \overline{x'}] \mapsto \bar{t}[y/x]^{[y:\overline{x'}]}) : t[y/x]^{[y:\overline{x'}]} \Downarrow ([y : \overline{x'}] +\!\!+ \overline{z}[y/x] \mapsto$
$\overline{u}[y/x]^{[y:\overline{x'}]}) : w[y/x]^{[y:\overline{x'}]} \Rightarrow$
$(\Gamma[y/x], \overline{x} \mapsto \bar{t}[y/x]^{\overline{x}}) : t[y/x]^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z}[y/x] \mapsto \overline{u}[y/x]^{\overline{x}}) : w[y/x]^{\overline{x}}$.

- SUBCASE: $x \in L$.
  Let $L' = L$.
  To prove: $\forall \overline{x} \notin L'$.
  $(\Gamma[y/x], \overline{x} \mapsto \overline{t}[y/x]^{\overline{x}}) : t[y/x]^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z}[y/x] \mapsto \overline{u}[y/x]^{\overline{x}}) : w[y/x]^{\overline{x}}$
  Let $\overline{x} \notin L' = L$.
  $\overline{x} \notin L \Rightarrow (\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}}$
  $\overline{x} \cap \{y\} = \emptyset$
  $\wedge y \notin \mathtt{names}(\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t) \cup \mathtt{names}((\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{y}}}) : w^{\overline{y}})$
  $= \mathtt{names}(\Gamma) \cup \mathtt{fv}(\overline{t}) \cup \mathtt{fv}(t) \cup \overline{y} \cup \overline{z} \cup \mathtt{fv}(\overline{u^{\overline{y}}}) \cup \mathtt{fv}(w^{\overline{y}})$
  $\overset{C1}{\Rightarrow} y \notin \mathtt{names}(\Gamma) \cup \mathtt{fv}(\overline{t}) \cup \mathtt{fv}(t) \cup \overline{y} \cup \overline{z} \cup \mathtt{fv}(\overline{u}) \cup \mathtt{fv}(w)$
  $\overset{C2}{\Rightarrow} y \notin \mathtt{names}(\Gamma) \cup \mathtt{fv}(\overline{t^{\overline{x}}}) \cup \mathtt{fv}(t^{\overline{x}}) \cup \overline{y} \cup \overline{z} \cup \mathtt{fv}(\overline{u^{\overline{x}}}) \cup \mathtt{fv}(w^{\overline{x}}) \cup \overline{x}$
  $\Rightarrow y \notin \mathtt{names}((\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}}) \cup \mathtt{names}((\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}})$.

  By induction hypothesis,
  $(\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}})[y/x] : (t^{\overline{x}})[y/x] \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}})[y/x] : (w^{\overline{x}})[y/x]$
  $\overset{x \cap \overline{x} = \emptyset}{\Rightarrow} (\Gamma[y/x], \overline{x} \mapsto \overline{t}[y/x]^{\overline{x}}) : t[y/x]^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z}[y/x] \mapsto \overline{u}[y/x]^{\overline{x}}) : w[y/x]^{\overline{x}}$.

CASE: $y \notin L$.
$\forall \overline{x} \notin L.(\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}}$,
$\Rightarrow \forall \overline{x} \notin L \cup \{y\}.(\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}}$.
Therefore we have now $y \in L \cup \{y\}$ and we are in the previous case.

□

6.8 Proof of Lemma 9 : LET_INTRO

Lemma 9

LET_INTRO $(\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}} \wedge \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ (\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t)$
$\Rightarrow \Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}}$.

Proof. We have to find a finite set L such that $\overline{x} \notin L$ and
$\forall \overline{y} \notin L.(\Gamma, \overline{y} \mapsto \overline{t^{\overline{y}}}) : t^{\overline{y}} \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{y}}}) : w^{\overline{y}}$.
Consider $L' = \mathtt{names}((\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}}) \cup \mathtt{names}((\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}})$.
By hypothesis, $(\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}}$.
Applying Lemma 8, $\forall \overline{y} \notin L'.(\Gamma, \overline{y} \mapsto \overline{t^{\overline{y}}}) : t^{\overline{y}} \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{y}}}) : w^{\overline{y}}$.
Let $L = L' \backslash \{\overline{x}\}$.
Therefore, $\forall \overline{y} \notin L.(\Gamma, \overline{y} \mapsto \overline{t^{\overline{y}}}) : t^{\overline{y}} \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{y}}}) : w^{\overline{y}}$.

□

The role of indirections in lazy natural semantics (extended version)

Technical Report 13/13
Lidia Sánchez-Gil[1], Mercedes Hidalgo-Herrero[2], and Yolanda Ortega-Mallén[1]

[1] Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain
[2] Facultad de Educación, Universidad Complutense de Madrid, Spain

Abstract. Launchbury defines a natural semantics for lazy evaluation and proposes an alternative call-by-name version which introduces indirections and does not update closures. These changes in the semantic rules are not so innocuous as they seem, so that the equivalence of both semantics is not straightforward. We separate the two modifications and define two intermediate semantics: one with indirections and the other without update. In the present work we focus on the introduction of indirections during β-reduction and study how the heaps, i.e., the sets of bindings, obtained with this kind of evaluation do relate with the heaps produced by substitution. As a heap represents the context of evaluation for a term, we first define an equivalence that identifies terms with the same meaning under a given context. This notion of *context* equivalence is extended to heaps. Finally, we define a relation between heap/term pairs to establish the equivalence between the alternative natural semantics and its corresponding version without indirections.

1 Motivation

Twenty years have elapsed since Launchbury first presented in [8] his natural semantics for lazy evaluation, a key contribution to the semantic foundation for non-strict functional programming languages like Haskell or Clean. Launchbury defines in [8] a natural semantics for lazy evaluation (*call-by-need*) where the set of *bindings*, i.e., (variable, expression) pairs, is explicitly managed to make possible their sharing. Throughout these years, Launchbury's natural semantics has been frequently cited and has inspired many further works as well as several extensions like in [2, 9, 15, 17]. In [12] the authors of this paper have extended the lambda calculus with a new expression that introduces parallelism when performing functional applications. *Parallel application* creates new processes to distribute the computation, and these distributed processes exchange values through communication channels. For that reason, we have presented an extension of Launchbury's natural semantics with parallel application. The success of Lanchbury's proposal lies in its simplicity. Expressions are evaluated with respect to a *context*, which is represented by a heap of *bindings*. This heap is explicitly managed to make possible the sharing of bindings, thus, modeling laziness.

In order to prove that this lazy (operational) semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury introduces some variations in the operational semantics. On the one hand, the update of bindings with their computed values is an operational notion without counterpart in the standard denotational semantics, so that the alternative natural semantics does no longer update bindings and becomes a *call-by-name* semantics. Moreover, in the alternative semantics self-references yield infinite reductions, while in the original semantics the reduction of a self-reference gets blocked. On the other hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach to this mechanism, applications are carried out in the alternative operational semantics by introducing *indirections*, i.e., variables bound to variables, instead of by performing the β-reduction through substitution.

Unfortunately, the proof of the equivalence between the natural semantics and its alternative version is detailed nowhere, and a simple induction turns out to be insufficient. Intuitively, both reduction systems

should lead to the same results. However, the *context-heap* semantics is too sensitive to the changes introduced by the alternative rules. Consequently, the equivalence cannot be directly established since final values may contain free variables that are dependent on the context of evaluation, which is represented by the heap of bindings. No updating leads to the duplication of bindings, and although these duplicated bindings, as well as the indirections, do not add relevant information to the context, it is awkward to prove this fact. Therefore, our challenge is to establish a way of relating the heaps obtained with each reduction system, and to prove that at the end the semantics are equivalent, so that any reduction of a term in one of the systems has its counterpart in the other. To facilitate this task we consider separately the no updating and the introduction of indirections, and we define two intermediate semantics.

In this paper we investigate the effect of introducing indirections in a setting without updates, and we analyze the similarities and differences between the reductions proofs obtained with and without indirections. This analysis provides a deep insight on the behavior of a context-heap semantics like Launchbury's.

We want to identify terms up to α-conversion, but dealing with α-equated terms usually implies the use of Barendregt's variable convention [3] to avoid the renaming of bound variables. However, the use of the variable convention is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [16]). Moreover, we intend to formalize our results with the help of the Coq [4] proof assistant. These reasons have led us to look for a system of binding amenable to formalization and we have chosen a *locally nameless* representation (as presented by Charguéraud in [6]). This is a mixed notation where bound variables names are replaced by de Bruijn indices [7], while free variables preserve their names. Although de Bruijn indices solve the problem with α-conversion, they make it very complicated to deal with heaps. Moreover, the formalization becomes unreadable. Hence, the mixed notation of naming variables is more convenient. A locally nameless version of Launchbury's natural semantics has been presented by the authors in [14] and [13].

The main contributions of the present work are:

1. An equivalence relation that identifies heaps that define the same free variables but whose corresponding closures may differ on *undefined free variables*;
2. A preorder that relates two heaps whenever the first can be transformed into the second by *eliminating indirections*;
3. An extension of the preorder relation for heap/term pairs expressing that two terms are equivalent if they have the same structure and their free variables, defined in the context of the respective heaps, are the same except for some indirections.
4. An equivalence theorem for Launchbury's alternative semantics and a version without indirections (and without update).

The paper is structured as follows: In Section 2, we review the lambda calculus and the two natural semantics described by Launchbury in [8]. We also introduce two intermediate semantics, each introducing just one alternative rule. In Section 3 we give a locally nameless representation of the calculus and the semantics presented in Section 2. In Section 4 we define equivalence and preorder relations on terms, heaps and also on heap/term pairs. We include a number of interesting results concerning these relations and, finally, we prove the equivalence of Launchbury's alternative semantics and the intermediate semantics without update and without indirections. In the last section we draw conclusions and outline our future work.

The proofs of theorems, propositions and lemmas are detailed in the Appendix.

2 Lazy natural semantics

In this section we review the natural semantics defined by Launchbury in [8] for lazy evaluation, together with the alternative rules for application and variables which transform the lazy semantics in a call-by-name semantics. In order to facilitate the comparison of these semantics, we focus independently on each change and define two intermediate semantics.

$$x \in \mathit{Var}$$
$$e \in \mathit{Exp} ::= x \mid \lambda x.e \mid (e\ x) \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e$$

Fig. 1. Restricted syntax of the extended λ-calculus

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^{n}) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \Downarrow \Delta : w}$$

Fig. 2. Natural semantics

2.1 Natural semantics

The language described in [8] is a normalized lambda calculus extended with recursive local declarations. The (restricted) abstract syntax appears in Figure 1. Normalization is achieved in two steps: First an α-conversion is performed so that bound variables have distinct names; in a second phase, arguments for applications are enforced to be variables. These static transformations simplify the definition of the operational rules.

The natural semantics defined by Launchbury in [8] follows a call-by-need strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : w$, that is, the expression e in the context of the heap Γ reduces to the value w in the context of the heap Δ. *Heaps* are partial functions from variables into expressions. Each pair (variable, expression) is called a *binding* and it is represented by $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings may be updated to their corresponding computed values. *Values* ($w \in \mathit{Val}$) are expressions in weak-head-normal-form (*whnf*). The semantic rules are shown in Figure 2. Despite of the normalization, in the VAR rule an α-conversion of the final value, represented as $\hat{w}$, is still needed to avoid name clashing. This renaming is justified by Barendregt's variable convention [3].

2.2 Alternative natural semantics

In order to prove the computational adequacy of the natural semantics (Figure 2) with respect to a standard denotational semantics, Launchbury introduces the alternative rules for application and variables shown in Figure 3. The AVAR rule removes update from the semantics. The effect of AAPP is the addition of an indirection $(y \mapsto x)$[1] instead of performing the β-reduction by substitution as in $e'[x/y]$ in APP. This increases the number of bindings in the heap.

In the following, the natural semantics (rules in Figure 2) is referred as the NS, and the alternative semantics (rules LAM, LET and those in Figure 3) as the ANS. We write $\Downarrow^A$ for reductions in the ANS.

Launchbury proves in [8] the correctness of the NS with respect to a standard denotational semantics, and a similar result for the ANS is easily obtained (as the authors of this paper have done in [11]). Therefore, the NS and the ANS are "denotationally" equivalent in the sense that if an expression is reducible (in some heap context) by both semantics then the obtained values have the same denotation. But this is insufficient for our purposes, because we want to ensure that if for some (heap : term) pair a reduction exists in any of the semantics, then there must exist a reduction in the other too and the final heaps must be related.

[1] Thanks to the normalization and the α-conversion $\hat{e}$ in AVAR, it is guaranteed that y is fresh in Θ.

$$\textsc{AVar}\quad \frac{(\Gamma, x \mapsto e) : \hat{e} \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow \Delta : w} \qquad \textsc{AApp}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad (\Theta, y \mapsto x) : e' \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

Fig. 3. Alternative rules

2.3 Two intermediate semantics

The changes introduced by the ANS might seem to involve no serious difficulties to prove that any reduction proof from a (heap : term) pair with the NS implies some reduction with the ANS, such that the final (heap : value) pairs are "equivalent", and vice versa. Unfortunately things are not so easy. On the one hand, the alternative rule for variables transforms the original call-by-need semantics into a call-by-name semantics because bindings are not updated and computed values are no longer shared. On the other hand, the addition of indirections also complicates the task of comparing the (heap : value) pairs obtained by each reduction system. Notice that the introduction of indirections produces larger heaps and the final values may depend on these aditional names.

To deal separately with these difficulties we introduce two intermediate semantics, each corresponding to the inclusion of just one of the modifications into the NS. Thus, the rules of the *Indirection Natural Semantics* (INS) are those of the NS (Figure 2) except for the application rule, that corresponds to the one in the alternative version, i.e., AApp in Figure 3. Analogously, the rules of the *No-update Natural Semantics* (NNS) are those of the NS but for the variable rule, that corresponds to the alternative AVar rule in Figure 3. We use $\Downarrow^I$ to represent reductions of the INS and $\Downarrow^N$ for those of the NNS. The following table summarizes the characteristics of the four reduction systems defined so far:

	NS	**INS**	**NNS**	**ANS**
Indirections	✗	✓	✗	✓
Update	✓	✓	✗	✗

3 A locally nameless representation

The syntax given in Figure 1 includes two name binders: λ-abstraction and `let`-declaration. The *named representation* requires a quotient structure respect to α-equivalence. To avoid this in our analysis, we opt for a *locally nameless representation* [6] that uses de Bruijn indices [7] to represent bound variables while names are retained for free variables. We have chosen such a mixed notation because heaps collect free variables whose names we are interested in preserving in order to identify them more easily.

As mentioned above, our locally nameless representation has already been presented in [14] and [13]. A complete definition with detailed explanations can be found there. Here we just show what is indispensable to understand the present work.

3.1 Locally nameless syntax

The locally nameless syntax corresponding to the lambda calculus of Figure 1 is shown in Figure 4. *Var* stands now for the set of *variables*, where *bound variables* and *free variables* are distinguished. When the language includes multibinders, Charguéraud proposes in [6] the use of two natural numbers to represent bound variables. We choose this notation to designate bound variables from `let`-declarations, which are in fact multibinders. The first number is a de Bruijn index that counts how many binders (abstraction or `let`) one needs to cross to the left to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind only one variable, so that the second number should be always zero to represent a correct term. In the following, a list like $\{t_i\}_{i=1}^n$ is represented as $\overline{t}$, with length $|\overline{t}| = n$.

$$
\begin{array}{lll}
x \in \mathit{Id} & & i, j \in \mathbb{N} \\
v \in \mathit{Var} & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in \mathit{LNExp} & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

Fig. 4. Locally nameless syntax

Computing the *free variables* of a term t, denoted by $\texttt{fv}(t)$, is very easy when using the locally nameless representation, since bound and free variables are syntactically different:

$$
\begin{array}{lll@{\qquad}lll}
\texttt{fv}(\texttt{bvar}\ i\ j) & = & \emptyset & \texttt{fv}(\texttt{fvar}\ x) & = & \{x\} \\
\texttt{fv}(\texttt{abs}\ t) & = & \texttt{fv}(t) & \texttt{fv}(\texttt{app}\ t\ v) & = & \texttt{fv}(t) \cup \texttt{fv}(v) \\
\texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = & \texttt{fv}(\overline{t}) \cup \texttt{fv}(t) & & &
\end{array}
$$

where $\texttt{fv}(\overline{t})$ collects the free variables of all the terms in a list $\overline{t}$.

A name $x \in \mathit{Id}$ is *fresh in a term* $t \in \mathit{LNExp}$, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t, i.e., $x \notin \texttt{fv}(t)$. This is extended to a list of names: $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t \stackrel{\texttt{def}}{=} \overline{x} \notin \texttt{fv}(t)$.

Definition 1. *Two terms $t, t' \in \mathit{LNExp}$ have the* same structure*, written $t \sim_S t'$, if they differ only in the names of their free variables:*

$$
\text{SS\_BVAR}\ \frac{}{(\texttt{bvar}\ i\ j) \sim_S (\texttt{bvar}\ i\ j)} \qquad \text{SS\_FVAR}\ \frac{}{(\texttt{fvar}\ x) \sim_S (\texttt{fvar}\ y)}
$$

$$
\text{SS\_ABS}\ \frac{t \sim_S t'}{(\texttt{abs}\ t) \sim_S (\texttt{abs}\ t')} \qquad \text{SS\_APP}\ \frac{t \sim_S t' \quad v \sim_S v'}{(\texttt{app}\ t\ v) \sim_S (\texttt{app}\ t'\ v')}
$$

$$
\text{SS\_LET}\ \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \sim_S \overline{t'} \quad t \sim_S t'}{(\texttt{let}\ \overline{t}\ \texttt{in}\ t) \sim_S (\texttt{let}\ \overline{t'}\ \texttt{in}\ t')}
$$

where $\overline{t} \sim_S \overline{t'} = \texttt{List.forall2}\ (\cdot \sim_S \cdot)\ \overline{t}\ \overline{t'}$. [2]

Next proposition states that $\sim_S$ is an equivalence relation on *LNExp*:

Proposition 1.

SS_REF $\quad t \sim_S t$

SS_SIM $\quad t \sim_S t' \Rightarrow t' \sim_S t$

SS_TRANS $\quad t \sim_S t' \wedge t' \sim_S t'' \Rightarrow t \sim_S t''$

Since there is no danger of name capture, *substitution* of variable names in a term is trivial in the locally nameless representation. We write $t[y/x]$ for replacing the occurrences of x by y in the term t.

Name substitution preserves the structure of a term:

Lemma 1.

SS_SUBST $\quad t[y/x] \sim_S t$

A *variable opening* operation is needed to manipulate locally nameless terms. This operation turns the outermost bound variables into free variables. The operation is defined in terms of a more general function with an extra parameter representing the nesting level of the binder to be open.

A term $t \in \mathit{LNExp}$ is *open with a list of names* $\overline{x} \subseteq \mathit{Id}$, written $t^{\overline{x}}$, as follows:

[2] We use an ML-like notation for operations on lists. For instance, `List.map` applies a function to every component in a list and `List.nth` refers to the nth-component. Elements in lists are numbered starting with 0 to match bound variables indices.

$$t^{\overline{x}} = \{0 \to \overline{x}\}t$$

where

$$\begin{array}{ll}
\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) & = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases} \\
\{k \to \overline{x}\}(\texttt{fvar}\ x) & = \texttt{fvar}\ x \\
\{k \to \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \to \overline{x}\}\ t) \\
\{k \to \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \to \overline{x}\}\ t)\ (\{k \to \overline{x}\}\ v) \\
\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \to \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}\ t) \\
\text{and } \{k \to \overline{x}\}\ \overline{t} & = \texttt{List.map}\ (\{k \to \overline{x}\}\cdot)\ \overline{t}.
\end{array}$$

For simplicity, we write t^x for the variable opening with a unitary list $[x]$. We illustrate this concept and its use with an example:

Example 1. Let $t \equiv \texttt{abs}\ (\texttt{let bvar 0 1}, \texttt{bvar 1 0 in app}\ (\texttt{abs bvar 2 0})\ (\texttt{bvar 0 1}))$. Hence, the body of the abstraction is:

$$u \equiv \texttt{let bvar 0 1}, \boxed{\texttt{bvar 1 0}}\ \texttt{in app}\ (\texttt{abs}\ \boxed{\texttt{bvar 2 0}})\ (\texttt{bvar 0 1}).$$

But then in u the bound variables referring to the outermost abstraction of t (shown squared) point to nowhere. The opening of u with variable x replaces with x the bound variables referring to an hypothetical binder with body u:
$u^x = \texttt{let bvar 0 1}, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar 0 1})$. □

In some occasions we are interested in applying the opening operation to a list of terms, so that we define $\overline{t}^{\overline{x}} = \texttt{List.map}\ (\cdot^{\overline{x}})\ \overline{t}$. From now on, $\overline{x}$ represents a list of pairwise-distinct names in Id.

The structure of a term that has been opened does not depend on the names chosen for the opening:

Lemma 2.
SS_OP $\quad |\overline{x}| = |\overline{y}| \Rightarrow t^{\overline{x}} \sim_S t^{\overline{y}}$

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct):

$${}^{\backslash\overline{x}}t = \{0 \leftarrow \overline{x}\}t$$

where

$$\begin{array}{ll}
\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & = \texttt{bvar}\ i\ j \\
\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) & = \begin{cases} \texttt{bvar}\ k\ j & \text{if } \exists j : 0 \leq j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise} \end{cases} \\
\{k \leftarrow \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}\ t) \\
\{k \leftarrow \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \leftarrow \overline{x}\}\ t)\ (\{k \leftarrow \overline{x}\}\ v) \\
\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}\ t) \\
\text{and } \{k \leftarrow \overline{x}\}\ \overline{t} & = \texttt{List.map}\ (\{k \leftarrow \overline{x}\}\ \cdot)\ \overline{t}.
\end{array}$$

Under some conditions variable closing and variable opening are inverse operations:

Lemma 3.
CLOSE_OPEN $\quad \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Leftrightarrow {}^{\backslash\overline{x}}(t^{\overline{x}}) = t$

For the other way around a condition on terms is required. The locally nameless syntax given in Figure 4 allows to build terms that have no corresponding expression in *Exp* (Figure 1). For instance, when bound variables indices are out of range. Those terms in *LNExp* that match expressions in *Exp* are called *locally-closed*, written $\texttt{lc}\ t$. The predicate lc is defined by the following rules:

$$\text{LC\_VAR}\ \frac{}{\texttt{lc}\ (\texttt{fvar}\ x)} \qquad \text{LC\_ABS}\ \frac{\forall x \notin L \subseteq Id \quad \texttt{lc}\ t^x}{\texttt{lc}\ (\texttt{abs}\ t)}$$

$$\text{LC\_APP}\ \frac{\texttt{lc}\ t \quad \texttt{lc}\ v}{\texttt{lc}\ (\texttt{app}\ t\ v)} \qquad \text{LC\_LET}\ \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id \quad \texttt{lc}\ [t : \overline{t}]^{\overline{x}}}{\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

where $\texttt{lc}\ \overline{t} = \texttt{List.forall}\ (\texttt{lc}\ \cdot)\ \overline{t}$ and $\overline{x}^n \notin L$ indicates that $\overline{x}$ is a list of n pairwise distinct names not belonging to the *(finite)* set L.

In the rules LC_ABS and LC_LET we use *cofinite quantification* as described in [1]. Cofinite quantification is an elegant alternative to "exist-fresh" conditions and provides stronger induction and inversion principles. We use the notation $[t : \overline{t}]$ to represent the list with head t and tail $\overline{t}$. Later on the empty list is represented as $[\,]$, a unitary list as $[t]$, and $+\!\!+$ stands for the concatenation of lists.

Now we can write down the property complementary of Lemma 3:

Lemma 4.
OPEN_CLOSE $\quad \texttt{lc}\ t \Rightarrow (\backslash^{\overline{x}} t)^{\overline{x}} = t$

The locally-closure condition of a term is independent of the names of its free variables:

Lemma 5.
SS_LC $\quad t \sim_S t' \wedge \texttt{lc}\ t \Rightarrow \texttt{lc}\ t'$

Moreover, any locally closed term can be expressed as the variable opening of another term that does not contain the names chosen for the opening:

Lemma 6.
LC_OP_VARS $\quad \texttt{lc}\ t \wedge \overline{x} \subseteq Id \Rightarrow \exists s \in \mathit{LNExp}.\, (\texttt{fresh}\ \overline{x}\ \texttt{in}\ s \wedge s^{\overline{x}} = t)$

This result is useful to express that a term depends on some set of names.

3.2 Locally nameless semantics

Bindings in a heap associate expressions to free variables, therefore bindings are now pairs $(\texttt{fvar}\ x, t)$ with $x \in Id$ and $t \in \mathit{LNExp}$. To simplify, we just write $x \mapsto t$. In the following, we represent a heap $\{x_i \mapsto t_i\}_{i=1}^n$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names that are defined in the heap:

$$\texttt{dom}(\emptyset) = \emptyset \qquad \texttt{dom}(\Gamma, x \mapsto t) = \texttt{dom}(\Gamma) \cup \{x\}$$

In a *well-formed* heap names are defined at most once and terms are locally closed. The predicate ok expresses that a heap is well-formed:

$$\text{OK-EMPTY}\ \frac{}{\texttt{ok}\ \emptyset} \qquad \text{OK-CONS}\ \frac{\texttt{ok}\ \Gamma \qquad x \notin \texttt{dom}(\Gamma) \qquad \texttt{lc}\ t}{\texttt{ok}\ (\Gamma, x \mapsto t)}$$

The function names returns the set of names that appear in a heap, i.e., the names occurring in the domain or in the terms in the right-hand side:

$$\texttt{names}(\emptyset) = \emptyset \qquad \texttt{names}(\Gamma, x \mapsto t) = \texttt{names}(\Gamma) \cup \{x\} \cup \texttt{fv}(t)$$

and this is used to define the freshness predicate of a list of names in a heap:

$$\texttt{fresh}\ \overline{x}\ \texttt{in}\ \Gamma = \overline{x} \notin \texttt{names}(\Gamma).$$

$$\text{LNLam}\quad \frac{\{\texttt{ok}\ \Gamma\} \qquad \{\texttt{lc}\ (\texttt{abs}\ t)\}}{\Gamma : \texttt{abs}\ t \Downarrow \Gamma : \texttt{abs}\ t}$$

$$\text{LNVar}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \cup \texttt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : \texttt{fvar}\ x \Downarrow (\Delta, x \mapsto w) : w}$$

$$\text{LNApp}\quad \frac{\Gamma : t \Downarrow \Theta : \texttt{abs}\ u \qquad \Theta : u^x \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin \texttt{dom}(\Delta)\}}{\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x) \Downarrow \Delta : w}$$

$$\text{LNLet}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{s^{\overline{x}}}) : w^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s^{\overline{x}}}) = \overline{s} \wedge {}^{\backslash \overline{x}}(w^{\overline{x}}) = w \quad \{\overline{y}^{|\overline{t}|} \notin L\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{s^{\overline{y}}}) : w^{\overline{y}}}$$

Fig. 5. Natural semantics with locally nameless representation

$$\text{ALNVar}\quad \frac{(\Gamma, x \mapsto t) : t \Downarrow \Delta : w}{(\Gamma, x \mapsto t) : \texttt{fvar}\ x \Downarrow \Delta : w}$$

$$\text{ALNApp}\quad \frac{\begin{array}{c}\Gamma : t \Downarrow \Theta : \texttt{abs}\ u \\ \forall y \notin L \subseteq Id \quad (\Theta, y \mapsto \texttt{fvar}\ x) : u^y \Downarrow ([y : \overline{z}] \mapsto \overline{s}^y) : w^y \ \wedge\ {}^{\backslash y}(\overline{s}^y) = \overline{s} \wedge {}^{\backslash y}(w^y) = w \\ \{x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin [z : \overline{z}]\} \qquad \{z \notin L\}\end{array}}{\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x) \Downarrow ([z : \overline{z}] \mapsto \overline{s}^z) : w^z}$$

Fig. 6. Alternative natural semantics with locally nameless representation

These definitions are extended to (heap : term) pairs:

$$\texttt{names}(\Gamma : t) = \texttt{names}(\Gamma) \cup \texttt{fv}(t) \qquad\qquad \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : t) = \overline{x} \notin \texttt{names}(\Gamma : t)$$

Substitution of variable names is extended to heaps as follows:

$$\emptyset[z/y] = \emptyset \qquad\qquad (\Gamma, x \mapsto t)[z/y] = (\Gamma[z/y], x[z/y] \mapsto t[z/y])$$
$$\text{where } x[z/y] = \begin{cases} z & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

It is required that $z \notin \texttt{dom}(\Gamma)$ to guarantee that name substitution preserves well-formedness.

The rules for the natural semantics (Figure 2) using the locally nameless representation are shown in Figure 5 and the alternative rules (Figure 3) in Figure 6. For clarity, side-conditions in the rules are written within braces to distinguish them from judgements.

Notice that we introduce cofinite quantification in rules LNLet and ALNApp. As it is explained in [6], the advantage of the cofinite rules over existential and universal ones is that the freshness side-conditions are not explicit. Nevertheless, the finite set L represents somehow the names that should be avoided during a reduction proof. We use the variable opening to express that the final heap and value may depend on the chosen names. For instance, in LNLet, $w^{\overline{x}}$ indicates that it depends on the names $\overline{x}$, but there is a common basis w (Lemma 6 is helpful to deal with this notation in proofs). Moreover, it is required that this basis does not contain occurrences of $\overline{x}$; this is is expressed by ${}^{\backslash \overline{x}}(w^{\overline{x}}) = w$. By contrast to the rules for the predicate `lc` (in Section 3.1), in the cofinite rules LNLet and ALNApp the names introduced in the (infinite) premises do appear in the conclusion too. Therefore, the conclusion has to be particularized to some selected names not belonging to L.

A more detailed explanation of the rules of Figure 5 can be found in [14]. New here are the alternative rules of Figure 6. The rule ALNVar is a direct translation of the rule AVar. Notice that the renaming of the term in the premise is no longer needed. The rule ALNApp follows a similar pattern to LNLet. The

cofinite quantification indicates that the name introduced in the heap for the indirection should be fresh. Among infinite valid candidates, the name z is chosen for the reduction. The list $\overline{z}$ represents the rest of names defined in the heap which is obtained after the reduction. Since the rhs terms of this final heap may refer to the name of the indirection, they are represented as $\overline{s}^z$, i.e., each rhs term is open with the name z (this can be done by Lemma 6). The condition $^{\backslash y}(\overline{s}^y) = \overline{s}$ ensures that s does not further depend on z. Similar considerations are valid for the final value.

3.3 Properties

The reduction systems corresponding to the rules given in Figures 5 and 6 verify a number of interesting properties. Since some of these properties are true for the four reduction systems defined in Section 2, in the following (and when not indicated otherwise) we use $\Downarrow^K$ to represent $\Downarrow$, $\Downarrow^A$, $\Downarrow^I$, and $\Downarrow^N$.

The first property is a *regularity* lemma that ensures that the judgements produced by the locally nameless rules involve only well-formed heaps and locally closed terms:

Lemma 7.

REGULARITY $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \wedge \mathtt{ok}\ \Delta \wedge \mathtt{lc}\ w$

Another general property is that definitions are not lost during reduction, i.e., heaps only can grow with new names:

Lemma 8.

DEF_NOT_LOST $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta).$

Moreover, in the case of no update (NNS and ANS), the bindings in the initial heap are preserved during the whole reduction:

Lemma 9.

NO_UPDATE $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \Gamma \subseteq \Delta$
where $\Downarrow^K$ *represents* $\Downarrow^N$ *and* $\Downarrow^A$

During reduction, names might be added to the heap by the rules LNLet and ALNApp. However, there is no "spontaneous generation" of names, i.e., any name occurring in a final (heap : value) pair must either appear already in the initial (heap : term) pair or be defined in the final heap:

Lemma 10.

ADD_NAMES $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \mathtt{names}(\Delta : w) \subseteq \mathtt{names}(\Gamma : t) \cup \mathtt{dom}(\Delta).$

The freshness of the names introduced by the rules LNLet and ALNApp is determined as follows:

Lemma 11.

NEW_NAMES1 $\quad \Gamma : t \Downarrow^N \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ \Gamma$
NEW_NAMES2 $\quad \Gamma : t \Downarrow^A \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ (\Gamma : t)$

The following *renaming* lemma ensures that the evaluation of a term is "independent" of the fresh names chosen during the reduction process. Furthermore, any name defined in the context heap can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reduction proofs for (heap : term) pairs are unique up to α-conversion of the names defined in the heap.

Lemma 12.

RENAMING1 $\quad \Gamma : t \Downarrow^K \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w)$
$\Rightarrow \Gamma[y/x] : t[y/x] \Downarrow^K \Delta[y/x] : w[y/x]$

RENAMING2 $\quad \Gamma : t \Downarrow^K \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w) \wedge x \notin \mathtt{dom}(\Gamma) \wedge x \in \mathtt{dom}(\Delta)$
$\Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$

Some of these properties are proved in [14] for $\Downarrow$.[3] The proofs are done by rule induction. In the Appendix we extend the proofs for the rules ALNVar and ALNApp.

[3] Actually, in [14] only RENAMING1 appears.

4 Indirections

The aim of this section is to prove the equivalence of NNS and ANS. Each semantics yields a different heap after evaluating the same (heap : term) pair, and it is necessary to analyze their differences, which lie in the introduced indirections. Recall that a heap represents the context for the evaluation of terms, so that we need to determine when two heaps represent "equivalent" evaluation contexts. More precisely, we show how *indirections*, i.e., bindings that just redirect to another variable name, can be removed while preserving the context represented by the heap.

An *indirection* is a binding of the form $x \mapsto \texttt{fvar}\ y$, that is, it just redirects to another variable name. The set of indirections of a given heap is defined as follows:

$$\texttt{Ind}(\emptyset) = \emptyset \qquad \texttt{Ind}(\Gamma, x \mapsto t) = \begin{cases} \texttt{Ind}(\Gamma) \cup \{x\} & \text{if } t \equiv \texttt{fvar}\ y \\ \texttt{Ind}(\Gamma) & \text{otherwise} \end{cases}$$

Obviously, $\texttt{Ind}(\Gamma) \subseteq \texttt{dom}(\Gamma)$.

The next example illustrates how these indirections are introduced in the heap during the evaluation of terms.

Example 2. Let us evaluate the term $t \equiv \texttt{app}\ (\texttt{abs}\ (\texttt{bvar}\ 0\ 0))\ x$, in the context $\Gamma = \{x \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\}$. Reductions obtained with the NNS and the ANS are:

$$\Gamma : t \Downarrow^N \{x \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)$$
$$\Gamma : t \Downarrow^A \{x \mapsto \texttt{abs}\ (\texttt{bvar}\ 0\ 0), y \mapsto \texttt{fvar}\ x\} : \texttt{abs}\ (\texttt{bvar}\ 0\ 0)$$

The value produced is exactly the same in both cases. However, when comparing the final heap in $\Downarrow^N$ with the final heap in $\Downarrow^A$, the latter contains an extra indirection, $y \mapsto \texttt{fvar}\ x$. This indirection corresponds to the binding introduced by the ALNApp rule to reduce the application in the term t.

The previous example gives us a hint of how to establish a relation between the heaps that are obtained with the NNS and those produced by the ANS. We want two heaps to be related if one can be obtained from the other by just eliminating some indirections. For this purpose we define how to remove indirections from a heap, while preserving the evaluation context represented by this heap. Erasing an indirection $x \mapsto \texttt{fvar}\ y$ from a heap implies not only the elimination of the binding itself, but also the substitution of y for x in the rest of bindings:

$$(\emptyset, x \mapsto \texttt{fvar}\ y) \ominus x = \emptyset \qquad ((\Gamma, z \mapsto t), x \mapsto \texttt{fvar}\ y) \ominus x = ((\Gamma, x \mapsto \texttt{fvar}\ y) \ominus x, z \mapsto t[y/x])$$

Since substitution preserves the structure of terms (Lemma 1), when erasing an indirection from a heap, the rest of indirections in the heap remain as indirections. Therefore, we generalize our definition to remove a list of indirections from a heap:

$$\Gamma \ominus [\,] = \Gamma \qquad \Gamma \ominus [x : \overline{x}] = (\Gamma \ominus x) \ominus \overline{x}$$

4.1 Context equivalence

In order to identify heaps, an equivalence relation in *LNHeap* must be defined. This relation is based on the equivalence of terms. The meaning of a term depends on the meaning of its free variables. However, if a free variable is undefined in the context of evaluation of a term, then the name of this free variable is irrelevant. Therefore, we consider that two terms are equivalent in a given context if they only differ in the names of the free variables that do not belong to the context.

Definition 2. *Let $V \subseteq Id$, and $t, t' \in LNExp$. We say that t and t' are* context-equivalent *in V, written $t \approx^V t'$, when:*

$$\text{CE-BVAR}\quad \frac{}{(\texttt{bvar}\ i\ j) \approx^V (\texttt{bvar}\ i\ j)} \qquad \text{CE-FVAR}\quad \frac{x, x' \notin V \vee x = x'}{(\texttt{fvar}\ x) \approx^V (\texttt{fvar}\ x')}$$

$$\text{CE-ABS}\quad \frac{t \approx^V t'}{(\texttt{abs}\ t) \approx^V (\texttt{abs}\ t')} \qquad \text{CE-APP}\quad \frac{t \approx^V t' \quad v \approx^V v'}{(\texttt{app}\ t\ v) \approx^V (\texttt{app}\ t'\ v')}$$

$$\text{CE-LET}\quad \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \approx^V \overline{t'} \quad t \approx^V t'}{(\texttt{let}\ \overline{t}\ \texttt{in}\ t) \approx^V (\texttt{let}\ \overline{t'}\ \texttt{in}\ t')}$$

where $\overline{t} \approx^V \overline{t'} = \texttt{List.forall2}\ (\cdot \approx^V \cdot)\ \overline{t}\ \overline{t'}$.

Fixed a set of names V, $\approx^V$ is an equivalence relation on *LNExp*:

Proposition 2.

CE_REF $t \approx^V t$

CE_SYM $t \approx^V t' \Rightarrow t' \approx^V t$

CE_TRANS $t \approx^V t' \wedge t' \approx^V t'' \Rightarrow t \approx^V t''$

If two terms are context-equivalent then they must have the same structure:

Lemma 13.

CE_SS $t \approx^V t' \Rightarrow t \sim_S t'$

Context-equivalence is preserved in smaller contexts, but also in contexts extended with fresh names:

Lemma 14.

CE_SUB $t \approx^V t' \wedge V' \subseteq V \Rightarrow t \approx^{V'} t'$

CE_ADD $t \approx^V t' \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ t' \Rightarrow t \approx^{V \cup \overline{x}} t'$

Under some conditions context-equivalence is preserved by variable opening and substitution:

Lemma 15.

CE_SUBS1 $t \approx^V t' \wedge x, y \notin V \Rightarrow t[y/x] \approx^V t'$

CE_SUBS2 $t \approx^V t' \wedge (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') \wedge x \in V \Rightarrow t[y/x] \approx^V t'[y'/x]$

CE_SUBS3 $t \approx^V t' \wedge y \notin V \wedge \texttt{fresh}\ y\ \texttt{in}\ t \wedge \texttt{fresh}\ y\ \texttt{in}\ t' \Rightarrow t[y/x] \approx^{V[y/x]} t'[y/x]$

CE_OP1 $t \approx^V t' \Rightarrow t^{\overline{x}} \approx^V t'^{\overline{x}}$

CE_OP2 $t \approx^V t' \wedge \overline{x}, \overline{y} \notin V \wedge |\overline{x}| = |\overline{y}| \Rightarrow t^{\overline{x}} \approx^V t'^{\overline{y}}$

CE_OP3 $t \approx^V t' \wedge (\texttt{fvar}\ x) \approx^V (\texttt{fvar}\ y) \Rightarrow t^x \approx^V t'^y$

Based on this equivalence on terms, we define a family of relations on (well-formed) heaps:

Definition 3. *Let $V \subseteq Id$, and $\Gamma, \Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-context-equivalent in V*, written $\Gamma \approx^V \Gamma'$, when:*

$$\text{HCE-EMPTY}\ \frac{}{\emptyset \approx^V \emptyset} \qquad \text{HCE-CONS}\ \frac{\Gamma \approx^V \Gamma' \quad t \approx^V t' \quad \texttt{lc}\ t \quad x \notin \texttt{dom}(\Gamma)}{(\Gamma, x \mapsto t) \approx^V (\Gamma', x \mapsto t')}$$

These relations are in fact equivalences for well-formed heaps:

Proposition 3.

HCE_REF $\texttt{ok}\ \Gamma \Rightarrow \Gamma \approx^V \Gamma$

HCE_SYM $\Gamma \approx^V \Gamma' \Rightarrow \Gamma' \approx^V \Gamma$

HCE_TRANS $\Gamma \approx^V \Gamma' \wedge \Gamma' \approx^V \Gamma'' \Rightarrow \Gamma \approx^V \Gamma''$

Moreover, when two heaps are heap-context-equivalent in some set of identifiers, then both are well-formed, have the same domain, and have the same indirections.

Lemma 16.
HCE_DOM $\quad \Gamma \approx^V \Gamma' \Rightarrow \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma')$
HCE_IND $\quad \Gamma \approx^V \Gamma' \Rightarrow \mathtt{Ind}(\Gamma) = \mathtt{Ind}(\Gamma')$
HCE_OK $\quad \Gamma \approx^V \Gamma' \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma'$

There is an alternative characterization for heap-context-equivalence which expresses that two heaps are context-equivalent whenever they have the same domain and each pair of corresponding bound terms is context-equivalent.

Lemma 17.
HCE_ALT $\quad \Gamma \approx^V \Gamma' \Leftrightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma' \wedge \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \wedge (x \mapsto t \in \Gamma \wedge x \mapsto t' \in \Gamma' \Rightarrow t \approx^V t')$

Next results guarantee uniqueness up to permutations of sequence of indirections that makes two heaps be equivalent. The order in which two indirections are removed from a heap can be exchanged and the produced heaps are context-equivalent.

Lemma 18.
HCE_SWAP $\quad \mathtt{ok}\ \Gamma \wedge x, y \in \mathtt{Ind}(\Gamma) \wedge x \neq y \Rightarrow \Gamma \ominus [x, y] \approx^{V-\{x,y\}} \Gamma \ominus [y, x]$

Considering context-equivalence on heaps, we are particularly interested in the case where the context coincides with the domain of the heaps:

Definition 4. *Let $\Gamma, \Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-equivalent*, written $\Gamma \approx \Gamma'$, if they are heap-context-equivalent in* $\mathtt{dom}(\Gamma)$*:*

$$\text{HE}\quad \frac{\Gamma \approx^{\mathtt{dom}(\Gamma)} \Gamma'}{\Gamma \approx \Gamma'}$$

Lemma 18 can be generalized to a list of indirections and its permutations.

Lemma 19.
HE_PERM $\quad \mathtt{ok}\ \Gamma \wedge \overline{x}, \overline{y} \subseteq \mathtt{Ind}(\Gamma) \Rightarrow (\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \Leftrightarrow \overline{y} \in \mathcal{S}(\overline{x}))$

where $\mathcal{S}(\overline{x})$ denotes the set of all permutations of $\overline{x}$.

4.2 Indirection Relation

Coming back to the idea, shown in Example 2, that a heap can be obtained from another by just removing some indirections, we define the following relation on heaps:

Definition 5. *Let $\Gamma, \Gamma' \in LNHeap$. Γ is* indirection-related *to Γ', written $\Gamma \gtrsim_I \Gamma'$, when:*

$$\text{IR-HE}\ \frac{\Gamma \approx \Gamma'}{\Gamma \gtrsim_I \Gamma'} \qquad \text{IR-IR}\ \frac{\mathtt{ok}\ \Gamma \quad \Gamma \ominus x \gtrsim_I \Gamma' \quad x \in \mathtt{Ind}(\Gamma)}{\Gamma \gtrsim_I \Gamma'}$$

There is an alternative characterization for the relation $\gtrsim_I$ which expresses that a heap is indirection-related to another whenever the later can be obtained from the former by erasing a sequence of indirections.

Proposition 4.
IR_ALT $\quad \Gamma \gtrsim_I \Gamma' \Leftrightarrow \mathtt{ok}\ \Gamma \wedge \exists\ \overline{x} \subseteq \mathtt{Ind}(\Gamma)\ .\ \Gamma \ominus \overline{x} \approx \Gamma'$

Furthermore, this sequence of indirections is unique up to permutations (by Lemma 19), and it corresponds to the difference between the domains of the related heaps:

Corollary 1.
IR_DOM_DOM $\quad \Gamma \gtrsim_I \Gamma' \Rightarrow \Gamma \ominus (\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')) \approx \Gamma'$ [4]

The *indirection-relation* is a preorder on the set of well-formed heaps:

Proposition 5.

IR_REF $\quad \mathtt{ok}\ \Gamma \Rightarrow \Gamma \succsim_I \Gamma$

IR_TRANS $\quad \Gamma \succsim_I \Gamma' \wedge \Gamma' \succsim_I \Gamma'' \Rightarrow \Gamma \succsim_I \Gamma''$

Additionally, the *indirection-relation* satisfies the following properties:

Lemma 20.

IR_DOM $\quad \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{dom}(\Gamma') \subseteq \mathtt{dom}(\Gamma)$

IR_IND $\quad \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{Ind}(\Gamma') \subseteq \mathtt{Ind}(\Gamma)$

IR_OK $\quad \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma'$

IR_DOM_HE $\quad \Gamma \succsim_I \Gamma' \wedge \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \Rightarrow \Gamma \approx \Gamma'$

IR_IR_HE $\quad (\Gamma \succsim_I \Gamma' \wedge \Gamma' \succsim_I \Gamma) \Leftrightarrow \Gamma \approx \Gamma'$

Since $\approx$ is an equivalence relation (Proposition 3), the set of well-formed heaps can be partitioned into mutually exclusive equivalence classes: $[\Gamma] = \{\Gamma' \in \mathit{LNHeap} \mid \Gamma \approx \Gamma'\}$. The *quotien set* is $\mathit{LNHeap}/\approx = \{[\Gamma] \mid \Gamma \in \mathit{LNHeap}\}$. The indirection-relation over heap-equivalence classes is defined as $[\Gamma] \succsim_I [\Gamma'] = \Gamma \succsim_I \Gamma'$. This definition is correct, i.e., it does not depend on the chosen representative of the class, and defines a partial order in $\mathit{LNHeap}/\approx$.

Lemma 21.

IREQ_HE_IREQ1 $\quad [\Gamma] \succsim_I [\Gamma'] \wedge \Delta \approx \Gamma \Rightarrow [\Delta] \succsim_I [\Gamma']$

IREQ_HE_IREQ2 $\quad [\Gamma] \succsim_I [\Gamma'] \wedge \Delta \approx \Gamma' \Rightarrow [\Gamma] \succsim_I [\Delta]$

Proposition 6.

IREQ_REF $\quad \mathtt{ok}\ \Gamma \Rightarrow [\Gamma] \succsim_I [\Gamma]$

IREQ_ANTSYM $\quad [\Gamma] \succsim_I [\Gamma'] \wedge [\Gamma'] \succsim_I [\Gamma] \Rightarrow [\Gamma] = [\Gamma']$

IREQ_TRANS $\quad [\Gamma] \succsim_I [\Gamma'] \wedge [\Gamma'] \succsim_I [\Gamma''] \Rightarrow [\Gamma] \succsim_I [\Gamma'']$

The *indirection-relation* can be extended to (heap : term) pairs.

Definition 6. *Let $\Gamma, \Gamma' \in \mathit{LNHeap}$, and $t, t' \in \mathit{LNExp}$. We say that $(\Gamma : t)$ is* indirection-related *to $(\Gamma' : t')$, written $(\Gamma : t) \succsim_I (\Gamma' : t')$, when*

$$\frac{\forall z \notin L \subseteq \mathit{Id} \quad (\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t')}{(\Gamma : t) \succsim_I (\Gamma' : t')}$$

Example 3. Let us consider the following heap and term:

$$\Gamma = \{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0)), y_0 \mapsto \mathtt{fvar}\ x_2\}$$
$$t\ = \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ \mathtt{bvar}\ 0\ 0)$$

The (heap : term) pairs related with $(\Gamma : t)$ by removing the sequences of indirections $[\,]$, $[y_0]$, $[x_0]$, and $[x_0, y_0]$ are, respectively, the following:

(*a*) $\{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0)), y_0 \mapsto \mathtt{fvar}\ x_2\}$
$\quad : \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0))$

(*b*) $\{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0))\}$
$\quad : \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0))$

(*c*) $\{x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0)), y_0 \mapsto \mathtt{fvar}\ x_2\}$
$\quad : \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0))$

(*d*) $\{x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0))\}$
$\quad : \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0))$

Notice that in Example 1 the (heap : term) pair obtained with the ANS is indirection-related to the pair obtained with the NNS, by just removing the indirection $y \mapsto \texttt{fvar}\ x$.

The indirection-relation over (heap : term) pairs satisfies the following properties:

Lemma 22.

IRHT_IRH $(\Gamma : t) \gtrsim_I (\Gamma' : t') \Rightarrow \Gamma \gtrsim_I \Gamma'$

IRHT_SS $(\Gamma : t) \gtrsim_I (\Gamma' : t') \Rightarrow t \sim_S t'$

IRHT_LC $(\Gamma : t) \gtrsim_I (\Gamma' : t') \Rightarrow \texttt{lc}\ t \wedge \texttt{lc}\ t'$

4.3 Equivalence

Now we are ready to establish the desired equivalence between the NNS and the ANS in the sense that if a reduction proof can be obtained with the ANS for some term in a given context heap, then there must exist a reduction proof in the NNS for that same (heap : term) pair such that the final (heap : value) is indirection-related to the final (heap : value) obtained with the ANS, and vice versa.

Theorem 1.

EQ_AN $\Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in \mathit{LNHeap} \,.\, \exists w_N \in \mathit{LNVal} \,.\, \Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \gtrsim_I (\Delta_N : w_N)$

EQ_NA $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in \mathit{LNHeap} \,.\, \exists w_A \in \mathit{LNVal} \,.\, \exists \overline{x} \subseteq \texttt{dom}(\Delta_N) - \texttt{dom}(\Gamma) \,.\, \exists \overline{y} \subseteq \mathit{Id} \,.\, |\overline{x}| = |\overline{y}| \wedge$
$\Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \gtrsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$

Notice that in the second part of the theorem, i.e., from NNS to ANS, a renaming may be needed. This renaming only affects names that are added to the heap during the reduction process. This is due to the fact that in the NNS names occurring in the evaluation term (that is t in the theorem) may disappear during the evaluation and, thus, may be chosen on some application of the rule LNLet and added to the final heap. This cannot happen in the ANS (NEW_NAMES2 in Lemma 11).

To prove this theorem by rule induction, a generalization is needed. Instead of evaluating the same term in the same initial heap, we consider indirection-related initial (heap : term) pairs:

Proposition 7.

EQ_IR_AN $(\Gamma_A : t_A) \gtrsim_I (\Gamma_N : t_N) \wedge$
$\forall \overline{x} \notin L \subseteq \mathit{Id} \,.\, \Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}} \wedge \ {}^{\backslash \overline{x}}(\overline{s}_A{}^{\overline{x}}) = \overline{s}_A \wedge \ {}^{\backslash \overline{x}}(w_A{}^{\overline{x}}) = w_A$
$\Rightarrow \exists \overline{y} \notin L \,.\, \exists \overline{s}_N \subset \mathit{LNExp} \,.\, \exists w_N \in \mathit{LNVal} \,.$
$\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}} \wedge \ {}^{\backslash \overline{z}}(\overline{s}_N{}^{\overline{z}}) = \overline{s}_N \wedge \ {}^{\backslash \overline{z}}(w_N{}^{\overline{z}}) = w_N \wedge \overline{z} \subseteq \overline{y}$
$\wedge ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$

EQ_IR_NA $(\Gamma_A : t_A) \gtrsim_I (\Gamma_N : t_N) \wedge$
$\forall \overline{x} \notin L \subseteq \mathit{Id} \,.\, \Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}} \wedge \ {}^{\backslash \overline{x}}(\overline{s}_N{}^{\overline{x}}) = \overline{s}_N \wedge \ {}^{\backslash \overline{x}}(w_N{}^{\overline{x}}) = w_N$
$\Rightarrow \exists \overline{z} \notin L \,.\, \exists \overline{s}_A \subset \mathit{LNExp} \,.\, \exists w_A \in \mathit{LNVal} \,.$
$\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}} \wedge \ {}^{\backslash \overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \wedge \ {}^{\backslash \overline{y}}(w_A{}^{\overline{y}}) = w_A \wedge \overline{z} \subseteq \overline{y}$
$\wedge ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$

Once again, cofinite quantification replaces freshness conditions. For instance, in EQ_IR_NA it is required that the names introduced during the reduction for the NNS do not collide with names that are already defined in the initial heap for the ANS. The cofinite quantification expresses that if there is an infinite number of "similar" reduction proofs for $(\Gamma_N : t_N)$, each introducing different names in the heap, one can chose a reduction proof such that the new bindings do not interfere with $(\Gamma_A : t_A)$.

Since there is no update in both ANS and NNS (see Lemma 9), a final heap is expressed as the initial heap plus some set of bindings, such as $(\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}})$. In this case, $\overline{x}$ represents the list of new names, i.e., those that have been added during the reduction of local declarations. Since the terms bound to these new

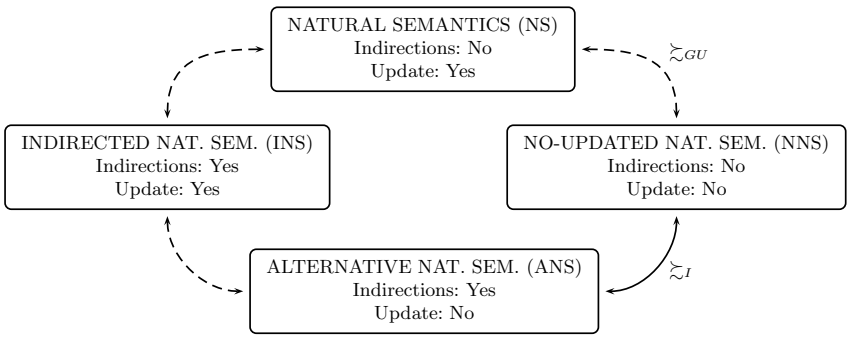


Fig. 7. The relations between the semantics

names are dependent on $\overline{x}$, they are represented as $\overline{s}_A{}^{\overline{x}}$. Similarly for the final value $w_A{}^{\overline{x}}$. The proposition indicates that it is possible to construct reductions for the NNS whose set of new defined names is a subset of the set of new names of the corresponding ANS reduction (which introduces new names when local declarations are evaluated by the rule LNLet, and also when indirections are created by the rule ALNApp).

5 Conclusions and Future Work

Launchbury natural semantics (NS) has turned out to be too much sensitive to the changes introduced by the alternative semantics (ANS), i.e., indirections and no-update. Intuitively, these changes should lead to the same values. However this cannot be directly established since values may contain free variables which are dependent on the context of evaluation, represented by the heap. And, precisely, the changes introduced by the ANS do affect deeply the heaps. In fact, the equivalence of the values produced by the NS and the ANS is based on their correctness with respect to a denotational semantics. Although indirections and duplicated bindings (consequence of the no-update) do not add new information to the heap, it is not straightforward to prove it formally.

Since the variations introduced by Launchbury in the ANS do affect two rules, i.e. the variable rule (no update) and the application rule (indirections), we have defined two intermediate semantics to deal separately with the effect of each modification: The NNS (without update) and the INS (with indirections). A schema of the semantics and how to related them is included in Figure 7.

In the present work we have compared the NNS with the ANS, that is, substitution vs. indirections. We have started by defining an equivalence $\approx$ such that two heaps are considered equivalent when they have the same domain and the corresponding closures may differ only in the free variables not defined in the heaps. We have used this equivalence to define a preorder $\succsim_I$ expressing that a heap can be transformed into another by eliminating indirections. Furthermore, the relation has been extended to (heap : terms) pairs, expressing that two terms can be considered equivalent when they have the same structure and their free variables (only those defined in the context of the corresponding heap) are the same except for some indirections. We have used this extended relation to establish the equivalence between the NNS and the ANS (Theorem 1).

Presently we are working on the equivalence of the NS and the NNS, which will close the path from the NS to the ANS. In order to compare the NS with the NNS, that is, update vs. no update, new relations on heaps and terms have to be defined. The no update of bindings in the heap corresponds in fact to a call-by-name strategy, and implies the duplication of evaluation work, that leads to the generation of duplicated bindings. These duplicated bindings come from several evaluations of the same `let`-declarations, so that they form *groups* of equivalent bindings. Therefore, we first define a preorder $\succsim_G$ that relates two

heaps whenever the first can be transformed into the second by eliminating duplicated groups of bindings. Afterwards, we define a relation $\sim_U$ that establishes when a heap is an updated version of another heap. Finally, both relations must be combined to obtain the *group-update* relation $\precsim_{GU}$, which extended for (heap : terms) will allow us to formulate an equivalence theorem for the NS and the NNS, similar to Theorem 1.

Although the relations $\precsim_I$ and $\precsim_{GU}$ are sufficient for proving the equivalence of the NS and the ANS, it would be interesting to complete the picture by comparing the NS with the INS, and then the INS with the AN. For the first step, we have to define a preorder similar to $\precsim_I$, but taking into account that extra indirections may now be updated, thus leading to "redundant" bindings. For the second step, some version of the group-update relation is needed. Dashed lines indicate this future work.

We have used a locally nameless representation to avoid problems with α-equivalence, while keeping a readable formalization of the syntax and semantics. This representation allow us to treat with heaps in a convenient and easy way, avoiding the problems that arise when using the de Bruijn notation. We have also introduced cofinite quantification (in the style of [1]) in the evaluation rules that introduce fresh names, namely the rule for local declarations (LNLet) and for the alternative application (ALNApp). Moreover, this representation is more amenable to formalization in proof assistants. In fact we have started to implement the semantic rules given in Section 3.2 using Coq [4], with the intention of obtaining a formal checking of our proofs. At this point we should mention the work of Joachim Breitner who is working on the formalization of the correctness and adequacy of Launchbury's semantics by using Isabelle [10]. So far he has been able to mechanically verify the correctness of the operational semantics with respect to the denotational one [5].

6 Acknowledgments

This work is partially supported by the projects: TIN2012-39391-C04-04 and S2009/TIC-1465.

References

1. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 3–15. ACM Press, 2008.
2. C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173. ACM Press, 2000.
3. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
4. Y. Bertot. Coq in a hurry. *CoRR*, abs/cs/0603118, 2006.
5. J. Breitner. The correctness of Launchbury's natural semantics for lazy evaluation. Archive of formal proofs, `http://afp.sf.net/entries/Launchbury`, 2013. Formal proof development.
6. A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.
7. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.
8. J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
9. K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.
10. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `http://isabelle.in.tum.de`.
11. L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. Call-by-need, call-by-name, and natural semantics. Technical Report UU-CS-2010-020, Departament of Information and Computing Sciences, Utrech University, 2010.
12. L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. *Trends in Functional Programming*, volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pages 65–80. Intellect, 2010.

13. L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. In *Theoretical Aspects of Computing (ICTAC 2012)*, pages 105–119. LNCS 7521, Springer, 2012.
14. L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation (extended version). Technical Report 01/12, Departamento de Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2012. http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-1-12.pdf.
15. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.
16. C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction (CADE-21)*, pages 35–50. LNCS 4603, Springer-Verlag, 2007.
17. M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud University Nijmegen, 2007.

7 Appendix

Some technical lemmas appear only in this appendix; since they are introduced before the results that need them, numeration is not consecutive. An ordered list can be found at the end of the appendix to facilitate the reading.

7.1 Results in Section 3.1

Proposition 1.

SS_REF $t \sim_S t$

SS_SIM $t \sim_S t' \Rightarrow t' \sim_S t$

SS_TRANS $t \sim_S t' \wedge t \sim_S t'' \Rightarrow t \sim_S t''$

Proof. *Reflexivity*, *symmetry* and *transitivity* are easily proved by induction on the structure of the terms. □

Lemma 1.

SS_SUBST $t[y/x] \sim_S t$

Proof. An easy structural induction on t. □

To prove Lemma 2 we need to prove first a more general result for terms that are opened at any level.

Lemma 23.

SS_OPK $t \sim_S t' \wedge |\overline{x}| = |\overline{y}| \Rightarrow \{k \to \overline{x}\}t \sim_S \{k \to \overline{y}\}t'$

Proof. By structural induction on t:

– $t \equiv \texttt{bvar}\ i\ j$:
$t \sim_S t' \Rightarrow t' \equiv \texttt{bvar}\ i\ j$.

$$\{k \mapsto \overline{x}\}(\texttt{bvar}\ i\ j) = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases}$$

$$\{k \mapsto \overline{y}\}(\texttt{bvar}\ i\ j) = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{y}) & \text{if } i = k \wedge j < |\overline{y}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases}$$

If $i = k$ and $j < |\overline{x}| = |\overline{y}|$, then $(\texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x})) \sim_S (\texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{y}))$, otherwise $(\texttt{bvar}\ i\ j) \sim_S (\texttt{bvar}\ i\ j)$.

- $t \equiv \texttt{fvar}\ x$:
  $t \sim_S t' \Rightarrow t' \equiv \texttt{fvar}\ y$.
  $\{k \mapsto \overline{x}\}(\texttt{fvar}\ x) = \texttt{fvar}\ x \sim_S \texttt{fvar}\ y = \{k \mapsto \overline{y}\}(\texttt{fvar}\ y)$.

- $t \equiv \texttt{abs}\ u$:
  $t \sim_S t' \Rightarrow t' \equiv \texttt{abs}\ u' \wedge u \sim_S u'$
  $\overset{I.H.}{\Rightarrow} \{k+1 \to \overline{x}\}u \sim_S \{k+1 \to \overline{y}\}u'$
  $\Rightarrow \texttt{abs}\ (\{k+1 \to \overline{x}\}u) \sim_S \texttt{abs}\ (\{k+1 \to \overline{y}\}u')$
  $\Rightarrow \{k \to \overline{x}\}(\texttt{abs}\ u) \sim_S \{k \to \overline{y}\}(\texttt{abs}\ u')$.
- $t \equiv \texttt{app}\ u\ v$:
  $t \sim_S t' \Rightarrow t' \equiv \texttt{app}\ u'\ v' \wedge u \sim_S u' \wedge v \sim_S v'$
  $\overset{I.H.}{\Rightarrow} \{k \to \overline{x}\}u \sim_S \{k \to \overline{y}\}u' \wedge \{k \to \overline{x}\}v \sim_S \{k \to \overline{y}\}v$
  $\Rightarrow \texttt{app}\ (\{k \to \overline{x}\}u)\ (\{k \to \overline{x}\}v) \sim_S \texttt{app}\ (\{k \to \overline{y}\}u')\ (\{k \to \overline{y}\}v')$
  $\Rightarrow \{k \to \overline{x}\}(\texttt{app}\ u\ v) \sim_S \{k \to \overline{y}\}(\texttt{app}\ u'\ v')$.
- $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ u$:
  $t \sim_S t' \Rightarrow t' \equiv \texttt{let}\ \overline{t'}\ \texttt{in}\ u' \wedge |\overline{t}| = |\overline{t'}| \wedge \overline{t} \sim_S \overline{t'} \wedge u \sim_S u'$
  $\overset{I.H.}{\Rightarrow} |\overline{t}| = |\overline{t'}| \wedge \{k+1 \to \overline{x}\}\overline{t} \sim_S \{k+1 \to \overline{y}\}\overline{t'} \wedge \{k+1 \to \overline{x}\}u \sim_S \{k+1 \to \overline{y}\}u'$
  $\Rightarrow \texttt{let}\ (\{k+1 \to \overline{x}\}\overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}u) \sim_S \texttt{let}\ (\{k+1 \to \overline{y}\}\overline{t'})\ \texttt{in}\ (\{k+1 \to \overline{y}\}u')$
  $\Rightarrow \{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ u) \sim_S \{k \to \overline{y}\}(\texttt{let}\ \overline{t'}\ \texttt{in}\ u')$.

□

Lemma 2.
SS_OP $\quad |\overline{x}| = |\overline{y}| \Rightarrow t^{\overline{x}} \sim_S t^{\overline{y}}$

Proof. By Proposition 1, $t \sim_S t$. A direct consequence of Lemma 23 (take $k = 0$). □

A proof for **Lemma 4** can be found in [14]. In that same technical report there is a proof for a part of **Lemma 3**, that is, $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow {}^{\backslash\overline{x}}(t^{\overline{x}}) = t$. We prove here the remaining result:

$$ {}^{\backslash\overline{x}}(t^{\overline{x}}) = t \Rightarrow \texttt{fresh}\ \overline{x}\ \texttt{in}\ t. $$

For this we need first to generalize to terms opened at any level.

Lemma 24.
OPK_CLK_FRESH $\quad \{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}t) = t \Rightarrow \texttt{fresh}\ \overline{x}\ \texttt{in}\ t$

Proof. By structural induction on t:

- $t \equiv \texttt{bvar}\ i\ j$: Immediate.
- $t \equiv \texttt{fvar}\ z$:
  $\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{fvar}\ z)) = \texttt{fvar}\ z$ only if $\forall j : 0 \leq j < |\overline{x}|.z \neq \texttt{List.nth}\ j\ \overline{x} \Rightarrow \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{fvar}\ z)$.
- $t \equiv \texttt{abs}\ u$:
  $\{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}(\texttt{abs}\ u)) = \{k \leftarrow \overline{x}\}(\texttt{abs}\ (\{k+1 \to \overline{x}\}u))$
  $= \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}(\{k+1 \to \overline{x}\}u)) = \texttt{abs}\ u$
  $\Rightarrow \{k+1 \leftarrow \overline{x}\}(\{k+1 \to \overline{x}\}u) = u$
  $\overset{I.H.}{\Rightarrow} \texttt{fresh}\ \overline{x}\ \texttt{in}\ u \Rightarrow \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{abs}\ u)$

The rest of cases are similar. □

Lemma 5.
SS_LC $\quad t \sim_S t' \wedge \texttt{lc}\ t \Rightarrow \texttt{lc}\ t'$

Proof. By structural induction on t:

– $t \equiv \texttt{abs}\ u$:
$\texttt{lc}\ t \quad \Rightarrow\ \forall x \notin L \subseteq Id.\texttt{lc}\ u^x.$
$t \sim_S t' \ \Rightarrow\ t' \equiv \texttt{abs}\ u' \wedge u \sim_S u' \overset{L23}{\Rightarrow} u^x \sim_S u'^x$ for any $x \in Id$ (by taking $\overline{x} = \overline{y} = [x]$ and $k = 0$)
$\overset{I.H.}{\Rightarrow} \forall x \notin L \subseteq Id.\texttt{lc}\ u'^x \Rightarrow \texttt{lc}\ (\texttt{abs}\ u')$

The rest of cases are either immediate or similar to the one shown above. □

To prove Lemma 6 we need a more general result for opening at an arbitrary level. The definition of *locally close at level k* can be found in [13].

Lemma 25.
LC_OPK_VARS $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \wedge \overline{x} \subseteq Id \Rightarrow \exists s \in LNExp.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ s \wedge \{k \to \overline{x}\}s = t)$

Proof. By structural induction on t:

– $t \equiv \texttt{bvar}\ i\ j$:
$\texttt{lc\_at}\ k\ \overline{n}\ \texttt{bvar}\ i\ j \Rightarrow i < k \Rightarrow \{k \mapsto \overline{x}\}(\texttt{bvar}\ i\ j) = \texttt{bvar}\ i\ j,$
and $\texttt{fresh}\ \overline{x}\ \texttt{in}\ \texttt{bvar}\ i\ j.$
Hence, take $s \equiv \texttt{bvar}\ i\ j$. Notice that $s \in Var$.
– $t \equiv \texttt{fvar}\ y$:
• $y \notin \overline{x}$: Take $s \equiv \texttt{fvar}\ y$.
• $y \in \overline{x}$: Take $s \equiv \texttt{bvar}\ k\ j$.
Notice that in both cases $s \in Var$.
– $t \equiv \texttt{abs}\ t'$:
$\texttt{lc\_at}\ k\ \overline{n}\ \texttt{abs}\ t' \Rightarrow \texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ t' \overset{I.H.}{\Rightarrow} \exists s'.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ s' \wedge \{k+1 \to \overline{x}\}s' = t').$
Take $s \equiv \texttt{abs}\ t'$, then $\{k \to \overline{x}\}s = \texttt{abs}\ (\{k+1 \to \overline{x}\}s') = \texttt{abs}\ t' = t.$
– $t \equiv \texttt{app}\ t'\ v$:
$\texttt{lc\_at}\ k\ \overline{n}\ \texttt{app}\ t'\ v \Rightarrow \texttt{lc\_at}\ k\ \overline{n}\ t' \wedge \texttt{lc\_at}\ k\ \overline{n}\ v$
$\overset{I.H.}{\Rightarrow} \exists s'.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ s' \wedge \{k \to \overline{x}\}s' = t') \wedge \exists v'.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ v' \wedge \{k \to \overline{x}\}v' = v).$
Take $s \equiv \texttt{app}\ t'\ v'$, then $\{k \to \overline{x}\}s = \texttt{app}\ (\{k \to \overline{x}\}s')\ (\{k \to \overline{x}\}v') = \texttt{app}\ t'\ v = t.$
– $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$:
$\texttt{lc\_at}\ k\ \overline{n}\ \texttt{let}\ \overline{t}\ \texttt{in}\ t' \Rightarrow \texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ [t' : \overline{t}]$
$\overset{I.H.}{\Rightarrow} \exists \overline{s}.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ \overline{s} \wedge \{k+1 \to \overline{x}\}\overline{s} = \overline{t}) \wedge \exists s'.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ s' \wedge \{k+1 \to \overline{x}\}s' = t').$
Take $s \equiv \texttt{let}\ \overline{s}\ \texttt{in}\ s'$, then $\{k \to \overline{x}\}s = \texttt{let}\ (\{k+1 \to \overline{x}\}\overline{s})\ \texttt{in}\ (\{k+1 \to \overline{x}\}s') = \texttt{let}\ \overline{t}\ \texttt{in}\ t' = t.$
□

Lemma 6.
LC_OP_VARS $\quad \texttt{lc}\ t \wedge \overline{x} \subseteq Id \Rightarrow \exists s \in LNExp.(\texttt{fresh}\ \overline{x}\ \texttt{in}\ s \wedge s^{\overline{x}} = t)$

Proof. Take $k = 0$ in Lemma 25. □

7.2 Results in Section 3.3

Remember that if not indicated otherwise $\Downarrow^K$ represents $\Downarrow$, $\Downarrow^A$, $\Downarrow^I$ and $\Downarrow^N$.

Lemma 7.
REGULARITY $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w$

Proof. This has been proved for $\Downarrow$ in [14] by rule induction. We just need to extend the induction proof for the alternative rules ALNVar and ALNApp:

– ALNVar
By induction hypothesis, $\texttt{ok}\ (\Gamma, x \mapsto t) \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w$.
By definition $\texttt{lc}\ (\texttt{fvar}\ x)$.

– ALNApp
By induction hypothesis, $\texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Theta \wedge \texttt{lc}\ (\texttt{abs}\ u)$ and
$\forall y \notin L.\texttt{ok}\ (\Theta, y \mapsto \texttt{fvar}\ x) \wedge \texttt{lc}\ u^y \wedge \texttt{ok}\ ([y : \overline{z}] \mapsto \overline{s}^y) \wedge \texttt{lc}\ w^y$.
Particularly for $z \notin L$, $\texttt{ok}\ ([z : \overline{z}] \mapsto \overline{s}^z) \wedge \texttt{lc}\ w^z$.
Since $\texttt{lc}\ t$ and $\texttt{lc}\ (\texttt{fvar}\ x)$, we have $\texttt{lc}\ (\texttt{app}\ t\ (\texttt{fvar}\ x))$ too.

□

Lemma 8.
DEF_NOT_LOST $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Delta)$

Proof. This has been proved for $\Downarrow$ in [14] by rule induction. We extend the proof for the alternative rules ALNVar and ALNApp:

– ALNVar:
By induction hypothesis, $\texttt{dom}(\Gamma, x \mapsto t) \subseteq \texttt{dom}(\Delta)$.

– ALNApp:
By induction hypothesis, $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Theta)$ and $\forall y \notin L.\texttt{dom}(\Theta, y \mapsto \texttt{fvar}\ x) \subseteq \texttt{dom}([y : \overline{z}] \mapsto \overline{s}^y)$.
Particularly for $z \notin L$, $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Theta) \subseteq \texttt{dom}(\Theta, z \mapsto \texttt{fvar}\ x) \subseteq \texttt{dom}([z : \overline{z}] \mapsto \overline{s}^z)$.

□

Lemma 9.
NO_UPDATE $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \Gamma \subseteq \Delta$
where $\Downarrow^K$ represents $\Downarrow^N$ and $\Downarrow^A$

Proof. A very easy rule induction. □

Lemma 10.
ADD_NAMES $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \texttt{names}(\Delta : w) \subseteq \texttt{names}(\Gamma : t) \cup \texttt{dom}(\Delta)$

Proof. This has been proved for $\Downarrow$ in [14] by rule induction. We extend the proof for the alternative rules ALNVar and ALNApp:

– ALNVar:

$$\begin{aligned}
\texttt{names}(\Delta : w) &\overset{I.H.}{\subseteq} \texttt{names}((\Gamma, x \mapsto t) : t) \cup \texttt{dom}(\Delta) \\
&= \texttt{names}(\Gamma, x \mapsto t) \cup \texttt{fv}(t) \cup \texttt{dom}(\Delta) \\
&= \texttt{names}(\Gamma, x \mapsto t) \cup \texttt{fv}(t) \cup \{x\} \cup \texttt{dom}(\Delta) \\
&= \texttt{names}((\Gamma, x \mapsto t) : \texttt{fvar}\ x) \cup \texttt{dom}(\Delta).
\end{aligned}$$

– ALNApp:
For any $y \notin L$:

$$\begin{aligned}
\texttt{names}(([y : \overline{z}] \mapsto \overline{s}^y) : w^y) &\overset{I.H.}{\subseteq} \texttt{names}((\Theta, y \mapsto \texttt{fvar}\ x) : u^y) \cup \texttt{dom}([y : \overline{z}] \mapsto \overline{s}^y) \\
&\overset{I.H.}{\subseteq} \texttt{names}(\Gamma : t) \cup \texttt{dom}(\Theta) \cup \{x\} \cup \texttt{dom}([y : \overline{z}] \mapsto \overline{s}^y) \\
&\overset{L8}{\subseteq} \texttt{names}(\Gamma : t) \cup \{x\} \cup \texttt{dom}([y : \overline{z}] \mapsto \overline{s}^y)
\end{aligned}$$

Particularly for $z \notin L$, $\texttt{names}(([z : \overline{z}] \mapsto \overline{s}^z) : w^z) \subseteq \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x)) \cup \texttt{dom}([z : \overline{z}] \mapsto \overline{s}^z)$.

□

To prove Lemma 11 we prove first a result concerning reductions of the ANS:

Lemma 26.
KEEP_NAMES $\quad \Gamma : t \Downarrow^A \Delta : w \Rightarrow \mathtt{fv}(t) \subseteq \mathtt{names}(\Delta : w)$

Proof. By rule induction:

- LNLam: Immediate.
- ALNVar: $\mathtt{fv}(\mathtt{fvar}\ x) = \{x\} \subseteq \mathtt{dom}(\Gamma, x \mapsto t) \overset{L8}{\subseteq} \mathtt{dom}(\Delta) \subseteq \mathtt{names}(\Delta : w)$.
- ALNApp:
  On the one hand, $\Gamma : t \Downarrow^A \Theta : \mathtt{abs}\ u \overset{I.H.}{\Rightarrow} \mathtt{fv}(t) \subseteq \mathtt{names}(\Theta : \mathtt{abs}\ u)$.
  But $\mathtt{names}(\Theta) \overset{L9}{\subseteq} \mathtt{names}([z : \overline{z}] \mapsto \overline{s}^z)$, and
  $\mathtt{fv}(\mathtt{abs}\ u) = \mathtt{fv}(u) \subseteq \mathtt{fv}(u^z) \overset{I.H.}{\subseteq} \mathtt{names}(([z : \overline{z}] \mapsto \overline{s}^z) : w^z)$.
  On the other hand, $x \in \mathtt{names}(\Theta, z \mapsto \mathtt{fvar}\ x) \overset{L9}{\subseteq} \mathtt{names}([z : \overline{z}] \mapsto \overline{s}^z)$.
  Therefore, $\mathtt{fv}(\mathtt{app}\ t\ (\mathtt{fvar}\ x)) \subseteq \mathtt{names}(([z : \overline{z}] \mapsto \overline{s}^z) : w^z)$.
- LNLet:
  $\mathtt{fv}(\overline{t}) \subseteq \mathtt{fv}(\overline{t}^{\overline{y}}) \subseteq \mathtt{names}(\Gamma, \overline{y} \mapsto \overline{t}^{\overline{y}}) \overset{L9}{\subseteq} \mathtt{names}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{y}})$, and
  $\mathtt{fv}(t) \subseteq \mathtt{fv}(t^{\overline{y}}) \overset{I.H.}{\subseteq} \mathtt{names}((\overline{y} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{y}}) : w^{\overline{y}})$.

□

Lemma 11.
NEW_NAMES1 $\quad \Gamma : t \Downarrow^N \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ \Gamma$
NEW_NAMES2 $\quad \Gamma : t \Downarrow^A \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ (\Gamma : t)$

Proof.
NEW_NAMES1
By rule induction:

- LNLam: Since $\Gamma = \Delta$, the result is immediate.
- ALNVar: Let $y \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma, x \mapsto t) \overset{I.H.}{\Rightarrow} \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma, x \mapsto t)$.
- LNApp: Let $y \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma)$:
  - $y \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Theta) \overset{I.H.}{\Rightarrow} \mathtt{fresh}\ y\ \mathtt{in}\ \Theta \overset{L9}{\Rightarrow} \mathtt{fresh}\ y\ \mathtt{in}\ \Gamma$.
  - $y \in \mathtt{dom}(\Theta) - \mathtt{dom}(\Gamma) \overset{I.H.}{\Rightarrow} \mathtt{fresh}\ y\ \mathtt{in}\ \Gamma$.
- LNLet: Let $x \in \mathtt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{y}}) - \mathtt{dom}(\Gamma)$:
  - $x \notin \overline{y} \Rightarrow x \notin \mathtt{dom}(\Gamma, \overline{y} \mapsto \overline{t}^{\overline{y}}) \overset{I.H.}{\Rightarrow} \mathtt{fresh}\ x\ \mathtt{in}\ (\Gamma, \overline{y} \mapsto \overline{t}^{\overline{y}}) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ \Gamma$.
  - $x \in \overline{y} \Rightarrow x \notin L$.
    Suppose that $x \in \mathtt{names}(\Gamma)$, hence, exists $z \mapsto t_z \in \Gamma$ such that $x \in \mathtt{fv}(t_z)$ for some $z \in \mathtt{dom}(\Gamma)$.
    Thus, $\forall \overline{x}^{|\overline{t}|} \notin L\,.\, {s_z}^{\overline{x}} = t_z$ and ${}^{\backslash \overline{x}}({s_z}^{\overline{x}}) = s_z$ for some fixed s_z.
    By Lemma 9, $s_z \in \overline{s}$ such that ${s_z}^{\overline{y}} = t_z$.
    If $x \notin \overline{x} \Rightarrow x \in \mathtt{fv}(s_z)$.
    If $x \in \overline{x} \overset{L3}{\Rightarrow} x \notin \mathtt{fv}(s_z)$.
    But this cannot be, since s_z is fixed.

NEW_NAMES2
By rule induction:

- LNLam: Since $\Gamma = \Delta$, the result is immediate.
- ALNVar: Let $y \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma, x \mapsto t) \overset{I.H.}{\Rightarrow} \mathtt{fresh}\ y\ \mathtt{in}\ ((\Gamma, x \mapsto t) : t)$.
  Moreover, $y \neq x$. Hence, $\mathtt{fresh}\ y\ \mathtt{in}\ ((\Gamma, x \mapsto t) : \mathtt{fvar}\ x)$.

– ALNApp: $t \equiv \texttt{app}\ t'\ (\texttt{fvar}\ x)$, and $\Delta = ([z : \overline{z}] \mapsto \overline{s}^z)$, which by Lemma 9 can be also expressed as $(\Gamma, z \mapsto \texttt{fvar}\ x, \overline{z}_1 +\!\!+ \overline{z}_2 \mapsto \overline{s}_1 +\!\!+ \overline{s}_2)$, with $\Gamma : t' \Downarrow^A (\Gamma, \overline{z}_1 \mapsto \overline{s}_1) : \texttt{abs}\ u$ and $(\Gamma, \overline{z}_1 \mapsto \overline{s}_1, z \mapsto \texttt{fvar}\ x) : u^z \Downarrow^A (\Gamma, \overline{z}_1 \mapsto \overline{s}_1, z \mapsto \texttt{fvar}\ x, \overline{z}_2 \mapsto \overline{s}_2) : w^z$.
Let $y \in \texttt{dom}(\Delta) - \texttt{dom}(\Gamma)$.
case 1: $y = z$.
Therefore, ${}^{\backslash y}(\overline{s}^y) = \overline{s} \wedge {}^{\backslash y}(w^y) = w \overset{L3}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ \overline{s} \wedge \texttt{fresh}\ y\ \texttt{in}\ w$
$\overset{y \notin \overline{z}}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ (([z' : \overline{z}] \mapsto \overline{s}^{z'}) : w^{z'}) = ((\Gamma, z' \mapsto \texttt{fvar}\ x, \overline{z}_1 +\!\!+ \overline{z}_2 \mapsto \overline{s}_1 +\!\!+ \overline{s}_2) : w^{z'})$ for all $z' \notin L \cup \{y\}$, and thus, $\texttt{fresh}\ y\ \texttt{in}\ \Gamma$.
Let us denote as $(\Delta' : w')$ the later (heap : value) pair,
$(\Gamma, \overline{z}_1 \mapsto \overline{s}_1, z' \mapsto \texttt{fvar}\ x) : u^{z'} \Downarrow^A \Delta' : w' \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Delta' : w') \overset{L26}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ u \wedge y \neq x$.
Furthermore, $\Gamma : t' \Downarrow^A (\Gamma, \overline{z}_1 \mapsto \overline{s}_1) : \texttt{abs}\ u \wedge \texttt{fresh}\ y\ \texttt{in}\ ((\Gamma, \overline{z}_1 \mapsto \overline{s}_1) : \texttt{abs}\ u) \overset{L26}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ t' \overset{y \neq x}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ \texttt{app}\ t'\ (\texttt{fvar}\ x)$.
case 2: $y \in \overline{z}_1$.
By induction hypothesis, $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t')$.
On the one hand, $x \in \texttt{dom}(\Gamma) \Rightarrow x \notin \overline{z_1} \Rightarrow x \neq y$;
on the other hand, by the rule $x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin [z : \overline{z}] \Rightarrow x \neq y$.
Therefore, $y \neq x$ at any case and $\texttt{fresh}\ y\ \texttt{in}\ \texttt{app}\ t'\ (\texttt{fvar}\ x)$.
case 3: $y \in \overline{z}_2$.
By induction hypothesis, $\texttt{fresh}\ y\ \texttt{in}\ ((\Gamma, \overline{z}_1 \mapsto \overline{s}_1, z \mapsto \texttt{fvar}\ x) : u^z) \Rightarrow \texttt{fresh}\ y\ \texttt{in}\ \Gamma$.
Also, $\texttt{fresh}\ y\ \texttt{in}\ ((\Gamma, \overline{z}_1 \mapsto \overline{s}_1, z \mapsto \texttt{fvar}\ x) : u^z) \Rightarrow y \neq x \wedge y \notin \texttt{names}((\Gamma, \overline{z}_1 \mapsto \overline{s}_1, z \mapsto \texttt{fvar}\ x) : \texttt{abs}\ u) \overset{L26}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ t'$.
Therefore, $\texttt{fresh}\ y\ \texttt{in}\ \texttt{app}\ t'\ (\texttt{fvar}\ x)$.
– LNLet: $t \equiv \texttt{let}\ \overline{t}\ \texttt{in}\ t'$, and let $x \in \texttt{dom}(\overline{y} +\!\!+ \overline{z} \mapsto \overline{s^y}) - \texttt{dom}(\Gamma)$.
- $x \notin \overline{y} \Rightarrow x \notin \texttt{dom}(\Gamma, \overline{y} \mapsto \overline{t}^{\overline{y}}) \overset{I.H.}{\Rightarrow} \texttt{fresh}\ x\ \texttt{in}\ ((\Gamma, \overline{y} \mapsto \overline{t}^{\overline{y}}) : t'^{\overline{y}}) \Rightarrow \texttt{fresh}\ x\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t')$.
- $x \in \overline{y} \Rightarrow x \notin L \wedge x \notin \overline{z}$.
  ${}^{\backslash \overline{y}}(\overline{s}^{\overline{y}}) = \overline{s} \wedge {}^{\backslash \overline{y}}(w^{\overline{y}}) = w \overset{L3}{\Rightarrow} \texttt{fresh}\ \overline{y}\ \texttt{in}\ \overline{s} \wedge \texttt{fresh}\ \overline{y}\ \texttt{in}\ w$
  $\Rightarrow \texttt{fresh}\ x\ \texttt{in}\ \overline{s} \wedge \texttt{fresh}\ x\ \texttt{in}\ w$
  $\Rightarrow \texttt{fresh}\ x\ \texttt{in}\ ((\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}}) : w^{\overline{x}})$, for all $\overline{x} \notin L \cup \{x\}$.
  Let us take any $\overline{x} \notin L \cup \{x\}$:
  On the one hand, $\texttt{names}(\Gamma) \subseteq \texttt{names}(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) \overset{L9}{\subseteq} \texttt{names}(\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}})$, and thus, $\texttt{fresh}\ x\ \texttt{in}\ \Gamma$.
  On the other hand, $(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t'^{\overline{x}} \Downarrow^A (\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}}) : w^{\overline{x}} \wedge \texttt{fresh}\ x\ \texttt{in}\ ((\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}}) : w^{\overline{x}})$
  $\overset{L26}{\Rightarrow} \texttt{fresh}\ x\ \texttt{in}\ \texttt{let}\ \overline{t}\ \texttt{in}\ t'$;

□

Lemma 12.

RENAMING1	$\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Delta : w)$ $\Rightarrow \Gamma[y/x] : t[y/x] \Downarrow^K \Delta[y/x] : w[y/x]$
RENAMING2	$\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Delta : w) \wedge x \notin \texttt{dom}(\Gamma) \wedge x \in \texttt{dom}(\Delta)$ $\Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$

Proof.
In [14] only RENAMING1 appears, but the proof of RENAMING2 is similar to the one of RENAMING1. We show here the proof cases (in the rule induction) for the alternative rules.

RENAMING1
We assume that $y \neq x$, otherwise the result is immediate.

– ALNVar:
$(\Gamma, z \mapsto t) : \texttt{fvar}\ z \Downarrow \Delta : w \Rightarrow (\Gamma, z \mapsto t) : t \Downarrow \Delta : w$;
$\texttt{fresh}\ y\ \texttt{in}\ ((\Gamma, z \mapsto t) : \texttt{fvar}\ z) \Rightarrow \texttt{fresh}\ y\ \texttt{in}\ ((\Gamma, z \mapsto t) : t)$,
so that by induction hypothesis, $(\Gamma[y/x], z[y/x] \mapsto t[y/x]) : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$,
that is, $(\Gamma[y/x], z' \mapsto t[y/x]) : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$, where $z' = z[y/x]$.
By rule ALNVar, $(\Gamma[y/x], z' \mapsto t[y/x]) : \texttt{fvar}\ z' \Downarrow \Delta[y/x] : w[y/x]$
$\Rightarrow (\Gamma, z \mapsto t)[y/x] : (\texttt{fvar}\ z)[y/x] \Downarrow \Delta[y/x] : w[y/x]$.

– ALNApp:
$\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x') \Downarrow ([z : \overline{z}] \mapsto \overline{s}^z) : w^z$
$\Rightarrow (x' \notin \texttt{dom}(\Gamma) \Rightarrow x' \notin [z : \overline{z}]) \wedge\ \Gamma : t \Downarrow \Theta : \texttt{abs}\ u$
$\wedge\ \forall y' \notin L.\ (\Theta, y' \mapsto \texttt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'}) : w^{y'} \wedge\ \backslash^y(\overline{s}^y) = \overline{s} \wedge\ \backslash^y(w^y) = w$,
for some finite $L \subseteq Id$ such that $z \notin L$.

$$\begin{array}{ll} \texttt{names}(\Gamma : t) & \subseteq\ \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x')) \\ \texttt{names}(\Theta : \texttt{abs}\ u) & \overset{L10}{\subseteq}\ \texttt{dom}(\Theta) \cup \texttt{names}(\Gamma : t) \subseteq \texttt{dom}(\Theta, z \mapsto \texttt{fvar}\ x') \cup \texttt{names}(\Gamma : t) \\ & \overset{L8}{\subseteq}\ \texttt{dom}([z : \overline{z}] \mapsto \overline{s}^z) \cup \texttt{names}(\Gamma : t) \\ & \subseteq\ \texttt{names}(([z : \overline{z}] \mapsto \overline{s}^z) : w^z) \cup \texttt{names}(\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x')). \end{array}$$

By hypothesis, $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x')) \wedge \texttt{fresh}\ y\ \texttt{in}\ (([z : \overline{z}] \mapsto \overline{s}^z) : w^z)$,
so that $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t)\ \wedge\ \texttt{fresh}\ y\ \texttt{in}\ (\Theta : \texttt{abs}\ u)$.
Therefore, by induction hypothesis,

$$\Gamma[y/x] : t[y/x] \Downarrow \Theta[y/x] : \underbrace{(\texttt{abs}\ u)[y/x]}_{\texttt{abs}\ u[y/x]} \tag{1}$$

Also, for any $y' \neq y$ we have $\texttt{fresh}\ y\ \texttt{in}\ ((\Theta, y' \mapsto \texttt{fvar}\ x') : u^{y'})\ \wedge\ \texttt{fresh}\ y\ \texttt{in}\ (([y' : \overline{z}] \mapsto \overline{s}^{y'}) : w^{y'})$,
and thus, by induction hypothesis,
$\forall y' \notin L \cup \{y\}.(\Theta, y' \mapsto \texttt{fvar}\ x')[y/x] : (u^{y'})[y/x] \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'})[y/x] : (w^{y'})[y/x]$.
This can be rewritten as

$$\forall y' \notin L'.(\Theta[y/x], y' \mapsto (\texttt{fvar}\ x')[y/x]) : u[y/x]^{y'} \Downarrow ([y' : \overline{z}[y/x]] \mapsto \overline{s}[y/x]^{y'}) : w[y/x]^{y'} \tag{2}$$

where $L' = \begin{cases} L \cup \{y\} & \text{if } x \in L \\ L \cup \{x\} - \{y\} & \text{if } x \notin L \end{cases}$ so that $z[y/x] \notin L'$.

Now we have to check that $\forall y' \notin L'.\backslash^y(\overline{s}[y/x]^y) = \overline{s}[y/x] \wedge\ \backslash^y(w^y) = w$.
Let $y' \notin L'$:

- $y' = y \Rightarrow x \notin L \Rightarrow\ \backslash^x(\overline{s}^x) = \overline{s} \overset{L3}{\Rightarrow} \texttt{fresh}\ x\ \texttt{in}\ \overline{s} \Rightarrow \overline{s}[y/x] = \overline{s}$;
  but $\texttt{fresh}\ y\ \texttt{in}\ \overline{s} \overset{L3}{\Rightarrow}\ \backslash^y(\overline{s}^y) = \overline{s}$.
- $y' \neq y \Rightarrow y' \notin L \Rightarrow\ \backslash^{y'}(\overline{s}^{y'}) = \overline{s} \overset{L3}{\Rightarrow} \texttt{fresh}\ y'\ \texttt{in}\ \overline{s} \Rightarrow \texttt{fresh}\ y'\ \texttt{in}\ \overline{s}[y/x] \overset{L3}{\Rightarrow}\ \backslash^{y'}(\overline{s}[y/x]^{y'}) = \overline{s}[y/x]$.

Similarly for w.

In order to apply ALNApp and obtain
$\Gamma[y/x] : \texttt{app}\ t[y/x]\ (\texttt{fvar}\ x'[y/x]) \Downarrow ([z[y/x] : \overline{z}[y/x]] \mapsto \overline{s}[y/x]^{z[y/x]}) : w[y/x]^{z[y/x]}$
from (1) and (2), we have to prove that if $x'[y/x] \notin \texttt{dom}(\Gamma[y/x])$ then $x'[y/x] \notin [z : \overline{z}][y/x]$:

- $x' = x \Rightarrow x'[y/x] = y$.
  Assume $y \notin \texttt{dom}(\Gamma[y/x]) \Rightarrow x' = x \notin \texttt{dom}(\Gamma) \overset{\text{hip.}}{\Rightarrow} x' = x \notin [z : \overline{z}]$.
  But $\texttt{fresh}\ y\ \texttt{in}\ ([z : \overline{z}] \mapsto \overline{s}^z)$, hence $y \notin [z : \overline{z}][y/x]$.
- $x' \neq x \Rightarrow x'[y/x] = x'$.
  Assume $x' \notin \texttt{dom}(\Gamma[y/x]) \Rightarrow x' \notin \texttt{dom}(\Gamma) \overset{\text{hip.}}{\Rightarrow} x' \notin [z : \overline{z}]$.
  Since $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x'))$ we have that $y \neq x'$. Therefore, $x' \notin [z : \overline{z}][y/x]$.

RENAMING2
We assume that $y \neq x$, otherwise the result is immediate.
By rule induction:

- ALNVAR:
  The proof is similar to the corresponding case in RENAMING1.

- ALNAPP:
  $\Rightarrow (x' \notin \mathtt{dom}(\Gamma) \Rightarrow x' \notin [z : \overline{z}]) \wedge\ \Gamma : t \Downarrow \Theta : \mathtt{abs}\ u$
  $\wedge\ \forall y' \notin L.\ (\Theta, y' \mapsto \mathtt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'}) : w^{y'} \wedge {}^{\backslash y}(\overline{s}^{y}) = \overline{s} \wedge {}^{\backslash y}(w^{y}) = w,$
  for some finite $L \subseteq Id$ such that $z \notin L$.

  Following the same steps as in RENAMING1 we obtain that $\mathtt{fresh}\ y\ \mathtt{in}\ (\Theta : \mathtt{abs}\ u)$, and that for any $y' \neq y$ it is $\mathtt{fresh}\ y\ \mathtt{in}\ ((\Theta, y' \mapsto \mathtt{fvar}\ x') : u^{y'})\ \wedge\ \mathtt{fresh}\ y\ \mathtt{in}\ (([y' : \overline{z}] \mapsto \overline{s}^{y'}) : w^{y'})$.
  Moreover, $\mathtt{fresh}\ y\ \mathtt{in}\ (([z : \overline{z}] \mapsto \overline{s}^{z}) : w^{z}) \Rightarrow y \neq z \wedge y \notin \overline{z}$,
  $\mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x')) \Rightarrow y \neq x'$, and
  $x \notin \mathtt{dom}(\Gamma) \wedge x \in \mathtt{dom}([z : \overline{z}] \mapsto \overline{s}^{z}) \Rightarrow x \neq x'$.

  CASE 1: $x \in \mathtt{dom}(\Theta)$
  On the one hand, by hypothesis $x \notin \mathtt{dom}(\Gamma)$, so that by induction hypothesis,

$$\Gamma : t \Downarrow \Theta[y/x] : \underbrace{(\mathtt{abs}\ u)[y/x]}_{\mathtt{abs}\ u[y/x]}$$

  On the other hand, if we define $L' = L \cup \{y\}$ we obtain

$$\forall y' \notin L'.(\Theta, y' \mapsto \mathtt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'}) : w^{y'}$$

  and by using RENAMING1 we have

$$\forall y' \notin L'.(\Theta, y' \mapsto \mathtt{fvar}\ x')[y/x] : u^{y'}[y/x] \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'})[y/x] : w^{y'}[y/x].$$

  As we are assuming $x \in \mathtt{dom}(\Theta)$, by Lemma 7 we have $\forall y' \notin L'.y' \neq x$; particularly, $z \neq x$. This fact, together with $x \neq x'$ allows to rewrite the last equation as

$$\forall y' \notin L'.(\Theta[y/x], y' \mapsto \mathtt{fvar}\ x') : u[y/x]^{y'} \Downarrow ([y' : \overline{z}][y/x] \mapsto \overline{s}[y/x]^{y'}) : w[y/x]^{y'}.$$

  Take any $y' \notin L' \Rightarrow y' \notin L \Rightarrow {}^{\backslash y}(\overline{s}^{y}) = \overline{s} \overset{L3}{\Rightarrow} \mathtt{fresh}\ y'\ \mathtt{in}\ \overline{s}\ \overset{y' \neq y}{\Rightarrow}\ \mathtt{fresh}\ y'\ \mathtt{in}\ \overline{s}[y/x] \overset{L3}{\Rightarrow} {}^{\backslash y}(\overline{s}[y/x]^{y}) = \overline{s}[y/x]$ (similarly for w).

  Furthermore, $z \notin L'$, and $x' \notin \mathtt{dom}(\Gamma) \Rightarrow x' \notin \mathtt{dom}([z : \overline{z}] \mapsto \overline{s}^{z})\ \overset{y \neq x'}{\Rightarrow}\ x' \notin \mathtt{dom}([z : \overline{z}][y/x] \mapsto \overline{s}[y/x]^{z})$.
  Then by rule ALNAPP,

$$\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x') \Downarrow ([z : \overline{z}][y/x] \mapsto \overline{s}[y/x]^{z}) : w[y/x]^{z}$$

  which, as $z \neq x$, can be rewritten to

$$\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x') \Downarrow ([z : \overline{z}] \mapsto \overline{s}^{z})[y/x] : w^{z}[y/x].$$

  CASE 2: $x \notin \mathtt{dom}(\Theta)$
  By hypothesis, $\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u$.
  We have to find a finite set $L' \subseteq Id$ such that $z[y/x] \notin L'$,
  $x' \notin \mathtt{dom}(\Gamma) \Rightarrow x' \notin \mathtt{dom}([z : \overline{z}][y/x] \mapsto \overline{s}^{z}[y/x])$, and

$$\forall y' \notin L'.(\Theta, y' \mapsto \mathtt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}[y/x]] \mapsto \overline{s}[y/x]^{y'}) : w[y/x]^{y'}$$

  and then apply the rule ALNAPP.

- SUBCASE 2.1: $x \neq z$.
  Let $L' = L \cup \{y\} \cup \{x\}$, then by induction hypothesis,

  $$\forall y' \notin L'.(\Theta, y' \mapsto \texttt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'})[y/x] : w^{y'}[y/x]$$

  which, as $y' \neq x$, can be rewritten to

  $$\forall y' \notin L'.(\Theta, y' \mapsto \texttt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}[y/x]] \mapsto \overline{s}[y/x]^{y'}) : w[y/x]^{y'}.$$

  Similarly as done in CASE 1, we check that ${}^{\backslash y'}(\overline{s}^{y'}) = \overline{s} \wedge {}^{\backslash y'}(w^{y'}) = w$, for all $y' \notin L'$.
  Furthermore, $z[y/x] \overset{z \neq x}{=} z \notin L \overset{z \neq y}{\Rightarrow} z[y/x] \notin L'$, and
  $x' \notin \texttt{dom}(\Gamma) \Rightarrow x' \notin \texttt{dom}([z : \overline{z}] \mapsto \overline{s}^z) \overset{x' \neq y}{\Rightarrow} x' \notin \texttt{dom}([z : \overline{z}[y/x]] \mapsto \overline{s}[y/x]^z)$.
  Hence, by rule ALNAPP,

  $$\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x') \Downarrow ([z[y/x] : \overline{z}[y/x]] \mapsto \overline{s}[y/x]^{z[y/x]}) : w[y/x]^{z[y/x]}$$

  that is $\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x') \Downarrow ([z : \overline{z}] \mapsto \overline{s}^z)[y/x] : w^z[y/x]$.
- SUBCASE 2.2: $x = z$.
  Therefore, ${}^{\backslash x}(\overline{s}^x) = \overline{s} \wedge {}^{\backslash x}(w^x) = w \overset{L3}{\Rightarrow} \texttt{fresh}\ x\ \texttt{in}\ \overline{s} \wedge \texttt{fresh}\ x\ \texttt{in}\ w$
  $\Rightarrow ([z : \overline{z}] \mapsto \overline{s}^z)[y/x] = ([x : \overline{z}] \mapsto \overline{s}^x)[y/x] = [y : \overline{z}] \mapsto \overline{s}^y$, and $w^z[y/x] = w^x[y/x] = w^y$.
  Furthermore, $x' \notin \texttt{dom}(\Gamma) \Rightarrow x' \notin \texttt{dom}([z : \overline{z}] \mapsto \overline{s}^z) \overset{x' \neq y}{\Rightarrow} x' \notin \texttt{dom}([y : \overline{z}] \mapsto \overline{s}^y)$.
  - $y \notin L$
    Then we are done, because by rule ALNAPP: $\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x') \Downarrow ([y : \overline{z}] \mapsto \overline{s}^y) : w^y$.
  - $y \in L$
    Let $L' = L \cup \{x\} \cup \{x'\} \cup \texttt{names}(\Gamma : t)$.
    Since $L' \subseteq L$ we have $\forall y' \notin L'.(\Theta, y' \mapsto \texttt{fvar}\ x') : u^{y'} \Downarrow ([y' : \overline{z}] \mapsto \overline{s}^{y'}) : w^{y'}$.
    Let us choose some $z' \notin L'$ such as $z' \neq y$.
    Hence, if $x' \notin \texttt{dom}(\Gamma) \Rightarrow x' \notin \texttt{dom}([z : \overline{z}] \mapsto \overline{s}^z) \overset{x' \neq z'}{\Rightarrow} x' \notin \texttt{dom}([z' : \overline{z}] \mapsto \overline{s}^{z'})$.
    Thus, by rule ALNAPP,

    $$\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x') \Downarrow ([z' : \overline{z}] \mapsto \overline{s}^{z'}) : w^{z'}.$$

    By hypothesis, $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x'))$, and
    $\texttt{fresh}\ y\ \texttt{in}\ (([z : \overline{z}] \mapsto \overline{s}^z) : w^z) \overset{y \neq z'}{\Rightarrow} \texttt{fresh}\ y\ \texttt{in}\ (([z' : \overline{z}] \mapsto \overline{s}^{z'}) : w^{z'})$
    So by RENAMING1 we obtain

    $$\Gamma[y/z'] : (\texttt{app}\ t\ (\texttt{fvar}\ x'))[y/z'] \Downarrow ([z' : \overline{z}] \mapsto \overline{s}^{z'})[y/z'] : w^{z'}[y/z'].$$

    Since $z' \notin \texttt{names}(\Gamma : t) \cup \{x'\}$, this is rewritten to $\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x') \Downarrow ([y : \overline{z}] \mapsto \overline{s}^y) : w^y$.

□

The following version of renaming will be useful in some cases.

Corollary 2.
RENAMING3 $\quad \Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Delta : w) \wedge x \notin \texttt{names}(\Gamma : t)$
$\Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$

Proof. If $x \notin \texttt{names}(\Gamma : t)$ then $(\Gamma[y/x] : t[y/x]) = (\Gamma : t)$. Thus, it is a corollary of RENAMING1. □

7.3 Results on context equivalence (Section 4.1)

Proposition 2.
CE_REF $t \approx^V t$
CE_SYM $t \approx^V t' \Rightarrow t' \approx^V t$
CE_TRANS $t \approx^V t' \wedge t' \approx^V t'' \Rightarrow t \approx^V t''$

Proof.
CE_REF and CE_SYM are proved easily by rule induction.

CE_TRANS
By structural induction on t:

- $t \equiv \texttt{fvar}\ y$.
  $t \approx^V t' \Rightarrow t' \equiv \texttt{fvar}\ y' \wedge (y, y' \notin V \vee y = y')$.
  - $y, y' \notin V$.
    $t' \approx^V t'' \Rightarrow t'' \equiv \texttt{fvar}\ y'' \wedge (y', y'' \notin V \vee y' = y'')$.
    * $y', y'' \notin V \Rightarrow y, y'' \notin V \Rightarrow (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y'')$.
    * $y' = y'' \Rightarrow y'' \notin V \Rightarrow (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y'')$.
  - $y = y'$. Immediate.

The rest of cases are very easy. □

Lemma 13.
CE_SS $t \approx^V t' \Rightarrow t \sim_S t'$

Proof. An easy structural induction on t. □

We add a corollary of the previous lemma which will be useful in forthcoming proofs.

Corollary 3.
CE_LC $t \approx^V t' \wedge \texttt{lc}\ t \Rightarrow \texttt{lc}\ t'$

Proof. By Lemmas 13 and 5. □

Lemma 14.
CE_SUB $t \approx^V t' \wedge V' \subseteq V \Rightarrow t \approx^{V'} t'$
CE_ADD $t \approx^V t' \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ t' \Rightarrow t \approx^{V \cup \overline{x}} t'$

Proof.

CE_SUB
By rule induction.
The only interesting case is CE-FVAR: $(\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') \Rightarrow y, y' \notin V \vee y = y'$.

- $y, y' \notin V \Rightarrow y, y' \notin V' \Rightarrow (\texttt{fvar}\ y) \approx^{V'} (\texttt{fvar}\ y')$.
- $y = y' \Rightarrow (\texttt{fvar}\ y) \approx^{V'} (\texttt{fvar}\ y')$.

CE_ADD
By rule induction.
The only interesting case is CE-FVAR: $(\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') \Rightarrow (y, y' \notin V \vee y = y')$.
$\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\texttt{fvar}\ y) \Rightarrow y \notin \overline{x}$, and similarly, $y' \notin \overline{x}$.
Therefore, $(\texttt{fvar}\ y) \approx^{V \cup \overline{x}} (\texttt{fvar}\ y')$. □

To prove Lemma 15 we prove first some results involving the operation of variable opening at level k.

Lemma 27.

CE_OPK1 $t \approx^V t' \Rightarrow \{k \to \overline{x}\}t \approx^V \{k \to \overline{x}\}t'$

CE_OPK2 $t \approx^V t' \wedge \overline{x}, \overline{y} \notin V \wedge |\overline{x}| = |\overline{y}| \Rightarrow \{k \to \overline{x}\}t \approx^V \{k \to \overline{y}\}t'$

Proof.

CE_OPK1
By rule induction:

- CE-ABS : $(\texttt{abs}\ t) \approx^V (\texttt{abs}\ t') \Leftrightarrow t \approx^V t'$.
  $\{k \to \overline{x}\}(\texttt{abs}\ t) = \texttt{abs}\ (\{k+1 \to \overline{x}\}t)$.
  $\{k \to \overline{x}\}(\texttt{abs}\ t') = \texttt{abs}\ (\{k+1 \to \overline{x}\}t')$.
  By induction hypothesis, $\{k+1 \to \overline{x}\}t \approx^V \{k+1 \to \overline{x}\}t'$.
  Hence, $\{k \to \overline{x}\}(\texttt{abs}\ t) \approx^V \{k \to \overline{x}\}(\texttt{abs}\ t')$.

The rest of cases are very easy.

CE_OPK2
By rule induction. The only interesting cases are the rules for bound and free variables:

- CE-BVAR : $(\texttt{bvar}\ i\ j) \approx^V (\texttt{bvar}\ i\ j)$.
  $\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases}$
  $\{k \to \overline{y}\}(\texttt{bvar}\ i\ j) = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{y}) & \text{if } i = k \wedge j < |\overline{y}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases}$
  Since $|\overline{x}| = |\overline{y}|$
  - either $\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) = \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x})$ and $\{k \to \overline{y}\}(\texttt{bvar}\ i\ j) = \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{y})$,
  - or $\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) = (\texttt{bvar}\ i\ j)$ and $\{k \to \overline{y}\}(\texttt{bvar}\ i\ j) = (\texttt{bvar}\ i\ j)$.

  Since $\overline{x}, \overline{y} \notin V$, the resulting terms are context equivalent in both cases.
- CE-FVAR : $(\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') \Leftrightarrow (y, y' \notin V \vee y = y')$.
  It is immediate, since $\{k \to \overline{x}\}(\texttt{fvar}\ z) = \texttt{fvar}\ z$.

□

Lemma 15.

CE_SUBS1 $t \approx^V t' \wedge x, y \notin V \Rightarrow t[y/x] \approx^V t'$

CE_SUBS2 $t \approx^V t' \wedge (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') \wedge x \in V \Rightarrow t[y/x] \approx^V t'[y'/x]$

CE_SUBS3 $t \approx^V t' \wedge y \notin V \wedge \texttt{fresh}\ y\ \texttt{in}\ t \wedge \texttt{fresh}\ y\ \texttt{in}\ t' \Rightarrow t[y/x] \approx^{V[y/x]} t'[y/x]$

CE_OP1 $t \approx^V t' \Rightarrow t^{\overline{x}} \approx^V t'^{\overline{x}}$

CE_OP2 $t \approx^V t' \wedge \overline{x}, \overline{y} \notin V \wedge |\overline{x}| = |\overline{y}| \Rightarrow t^{\overline{x}} \approx^V t'^{\overline{y}}$

CE_OP3 $t \approx^V t' \wedge (\texttt{fvar}\ x) \approx^V (\texttt{fvar}\ y) \Rightarrow t^x \approx^V t'^y$

Proof.

CE_SUBS1
An easy rule induction.

CE_SUBS2
By rule induction:

- CE-FVAR : $(\texttt{fvar}\ z) \approx^V (\texttt{fvar}\ z') \Rightarrow (z, z' \notin V \vee z = z')$.
  CASE 1: $z, z' \notin V \Rightarrow z \neq x \wedge z' \neq x$.
  Hence, $(\texttt{fvar}\ z)[y/x] = (\texttt{fvar}\ z)$ and $(\texttt{fvar}\ z')[y/x] = (\texttt{fvar}\ z')$.
  CASE 2: $z = z'$
  - $z = x \Rightarrow z' = x$
    $(\texttt{fvar}\ z)[y/x] = (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') = (\texttt{fvar}\ z')[y'/x]$.

- $z \neq x \Rightarrow z' \neq x$. Similar to CASE 1.

The rest of cases are immediate.

CE_SUBS3
By rule induction:

- CE-FVAR : $(\texttt{fvar}\ z) \approx^V (\texttt{fvar}\ z') \Rightarrow (z, z' \notin V \vee z = z')$.
  CASE 1: $z, z' \notin V$
  - $z = x \Rightarrow V[y/x] = V \wedge (\texttt{fvar}\ z)[y/x] = \texttt{fvar}\ y \wedge y \notin V \Rightarrow (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ z')$.
  - $z' = x$. Analogous.
  - $z \neq x \wedge z' \neq x \Rightarrow (\texttt{fvar}\ z)[y/x] = \texttt{fvar}\ z \wedge (\texttt{fvar}\ z')[y/x] = \texttt{fvar}\ z'$.
    And $z, z' \notin V \wedge y \neq z \wedge y \neq z' \Rightarrow z, z' \notin V[y/x]$.

  CASE 2: $z = z'$. Immediate.

The rest of cases are immediate.

CE_OP1
Take $k = 0$ in CE_OPK1 (Lemma 27).

CE_OP2
Take $k = 0$ in CE_OPK2 (Lemma 27).

CE_OP3
Since $(\texttt{fvar}\ x) \approx^V (\texttt{fvar}\ x')$, either

- $x = y$, and we use CE_OP1; or,
- $x, y \notin V$, and we use CE_OP2.

□

Lemma 16.

HCE_DOM	$\Gamma \approx^V \Gamma' \Rightarrow \texttt{dom}(\Gamma) = \texttt{dom}(\Gamma')$
HCE_IND	$\Gamma \approx^V \Gamma' \Rightarrow \texttt{Ind}(\Gamma) = \texttt{Ind}(\Gamma')$
HCE_OK	$\Gamma \approx^V \Gamma' \Rightarrow \texttt{ok}\ \Gamma \wedge \texttt{ok}\ \Gamma'$

Proof.
The three properties are proved by rule induction, where the case of the empty heap is immediate.

HCE_DOM
$\Gamma = (\Theta, y \mapsto t) \approx^V \Gamma' = (\Theta', y \mapsto t')$ with $\Theta \approx^V \Theta' \wedge y \notin \texttt{dom}(\Theta)$.
$\texttt{dom}(\Gamma) = \texttt{dom}(\Theta, y \mapsto t) = \texttt{dom}(\Theta) \cup \{y\} \overset{I.H.}{=} \texttt{dom}(\Theta') \cup \{y\} = \texttt{dom}(\Theta', y \mapsto t') = \texttt{dom}(\Gamma')$.

HCE_IND
$\Gamma = (\Theta, y \mapsto t) \approx^V \Gamma' = (\Theta', y \mapsto t')$ with $\Theta \approx^V \Theta' \wedge t \approx^V t' \wedge y \notin \texttt{dom}(\Theta)$.

$$\texttt{Ind}(\Theta, y \mapsto t) = \begin{cases} \texttt{Ind}(\Theta) \cup \{y\} & \text{if } t \equiv \texttt{fvar}\ z \\ \texttt{Ind}(\Theta) & \text{otherwise} \end{cases}$$

$$\texttt{Ind}(\Theta', y \mapsto t') = \begin{cases} \texttt{Ind}(\Theta') \cup \{y\} & \text{if } t' \equiv \texttt{fvar}\ z' \\ \texttt{Ind}(\Theta') & \text{otherwise} \end{cases}$$

But $t \approx^V t' \overset{L13}{\Rightarrow} t \sim_S t'$, therefore, $t \equiv \texttt{fvar}\ z \Leftrightarrow t' \equiv \texttt{fvar}\ z'$.

HCE_OK
$\Gamma = (\Theta, y \mapsto t) \approx^V \Gamma' = (\Theta', y \mapsto t')$ with $\Theta \approx^V \Theta' \wedge t \approx^V t' \wedge \texttt{lc}\ t \wedge y \notin \texttt{dom}(\Theta)$.

$\Theta \approx^V \Theta' \overset{I.H.}{\Rightarrow} \mathtt{ok}\ \Theta \wedge \mathtt{ok}\ \Theta'$.
$\mathtt{lc}\ t \overset{C3}{\Rightarrow} \mathtt{lc}\ t'$.
$y \notin \mathtt{dom}(\Theta) \overset{\text{HCE\_DOM}}{=} \mathtt{dom}(\Theta')$.
Thus, $\mathtt{ok}\ (\Theta, y \mapsto t)$ and $\mathtt{ok}\ (\Theta', y \mapsto t')$. □

To prove that $\approx^V$ is an equivalence on heaps, we prove first that the relation is independent of the order in which the bindings of the heaps are considered. So that when two heaps are related (in some context), all the names defined in those heaps are related.

Lemma 28.
HCE_BIND $\quad (\Gamma, x \mapsto t) \approx^V (\Gamma', x \mapsto t') \Rightarrow \Gamma \approx^V \Gamma' \wedge t \approx^V t'$

Proof. By induction on the size of Γ.

- $\Gamma = \emptyset$.
  $(\emptyset, x \mapsto t) \approx^V (\Gamma', x \mapsto t') \Rightarrow \Gamma = \emptyset \approx^V \Gamma' \wedge t \approx^V t' \wedge \mathtt{lc}\ t$.

- $\Gamma \neq \emptyset$.
  Let $\Delta = (\Gamma, x \mapsto t)$ and $\Delta' = (\Gamma', x \mapsto t')$.
  $\Delta \approx^V \Delta' \Rightarrow \exists y.\ \Delta = (\Theta, y \mapsto t_y) \wedge \Delta' = (\Theta', y \mapsto t'_y) \wedge \Theta \approx^V \Theta' \wedge t_y \approx^V t'_y \wedge \mathtt{lc}\ t_y$.
  If $y = x$ then by HCE_OK in Lemma 16 there is a unique x in Δ and Δ', so that $\Gamma = \Theta \wedge \Gamma' = \Theta'$, and we are done.
  Otherwise, $\Theta = (\Theta_x, x \mapsto t)$ and $\Theta' = (\Theta'_x, x \mapsto t')$, being $(\Theta_x, y \mapsto t_y) = \Gamma$, and hence $\Theta_x \subsetneq \Gamma$.
  By induction hypotesis, $\Theta_x \approx^V \Theta'_x \wedge t \approx^V t' \wedge \mathtt{lc}\ t$.
  Furthermore, $\Theta_x \approx^V \Theta'_x \wedge t_y \approx^V t'_y \wedge \mathtt{lc}\ t_y \Rightarrow \Gamma = (\Theta_x, y \mapsto t_y) \approx^V (\Theta'_x, y \mapsto t'_y) = \Gamma'$.

□

Proposition 3
HCE_REF $\quad \mathtt{ok}\ \Gamma \Rightarrow \Gamma \approx^V \Gamma$
HCE_SYM $\quad \Gamma \approx^V \Gamma' \Rightarrow \Gamma' \approx^V \Gamma$
HCE_TRANS $\quad \Gamma \approx^V \Gamma' \wedge \Gamma' \approx^V \Gamma'' \Rightarrow \Gamma \approx^V \Gamma''$

Proof.
HCE_REF and HCE_SYM are immediate.

HCE_TRANS
By rule induction on $\Gamma \approx^V \Gamma'$:

- HCE_EMPTY. $\emptyset \approx^V \emptyset \Rightarrow \Gamma'' = \emptyset$. Immediate.

- HCE_CONS. $\Gamma = (\Theta, y \mapsto t) \approx^V \Gamma' = (\Theta', y \mapsto t') \Rightarrow \Theta \approx^V \Theta' \wedge t \approx^V t' \wedge \mathtt{lc}\ t \wedge y \notin \mathtt{dom}(\Theta)$
  $\Gamma' \approx^V \Gamma'' \overset{L16}{\Rightarrow} \mathtt{dom}(\Gamma') = \mathtt{dom}(\Gamma'') \Rightarrow \Gamma'' = (\Theta'', y \mapsto t'') \overset{L28}{\Rightarrow} \Theta' \approx^V \Theta'' \wedge t' \approx^V t''$.
  By induction hypothesis, $\Theta \approx^V \Theta''$.
  By CE_TRANS in Proposition 2, $t \approx^V t''$.
  Hence, $\Gamma = (\Theta, y \mapsto t) \approx^V (\Theta'', y \mapsto t'') = \Gamma''$.

□

Lemma 17
HCE_ALT $\quad \Gamma \approx^V \Gamma' \Leftrightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma' \wedge \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \wedge (x \mapsto t \in \Gamma \wedge x \mapsto t' \in \Gamma' \Rightarrow t \approx^V t')$

Proof.
$\Rightarrow$ By HCE_DOM and HCE_OK (Lemma 16) and HCE_BIND (Lemma 28).

$\Leftarrow$ By induction on the size of Γ:

- $\Gamma = \emptyset$. Immediate.
- $\Gamma = (\Theta, y \mapsto t_y) \Rightarrow \Gamma' = (\Theta', y \mapsto t'_y) \wedge \mathtt{dom}(\Theta) = \mathtt{dom}(\Theta') \wedge t_y \approx^V t'_y$.
  $\mathtt{ok}\ \Gamma \Rightarrow \mathtt{ok}\ \Theta \wedge y \notin \mathtt{dom}(\Theta) \wedge \mathtt{lc}\ t_y$,
  $\mathtt{ok}\ \Gamma' \Rightarrow \mathtt{ok}\ \Theta' \wedge y \notin \mathtt{dom}(\Theta') \wedge \mathtt{lc}\ t'_y$.
  If $x \mapsto t \in \Theta \subseteq \Gamma \wedge x \mapsto t' \in \Theta' \subseteq \Gamma'$ then $t \approx^V t'$.
  Hence $\mathtt{ok}\ \Theta \wedge \mathtt{ok}\ \Theta' \wedge \mathtt{dom}(\Theta) = \mathtt{dom}(\Theta') \wedge (x \mapsto t \in \Theta \wedge x \mapsto t' \in \Theta' \Rightarrow t \approx^V t')$.
  By induction hypothesis, $\Theta \approx^V \Theta'$.
  Thus, $\Gamma = (\Theta, y \mapsto t_y) \approx^V (\Theta', x \mapsto t'_y) = \Gamma'$.

□

The following auxiliary result is used in the proofs of the Lemmas 18 and 19:

Lemma 29.
IND_DOM $\quad \mathtt{ok}\ \Gamma \wedge x \in \mathtt{Ind}(\Gamma) \Rightarrow \mathtt{dom}(\Gamma \ominus x) = \mathtt{dom}(\Gamma) - \{x\} \wedge \mathtt{Ind}(\Gamma \ominus x) = \mathtt{Ind}(\Gamma) - \{x\}$

Proof.
An easy induction on the size of Γ. □

We also give here some other properties related to the deletion of indirections. These will be used in forthcoming proofs.

Lemma 30.
IND_OK $\quad \mathtt{ok}\ \Gamma \wedge x \in \mathtt{Ind}(\Gamma) \Rightarrow \mathtt{ok}\ (\Gamma \ominus x)$

Proof. $\Gamma = (\Theta, x \mapsto \mathtt{fvar}\ y)$.
$\mathtt{ok}\ \Gamma \Rightarrow \mathtt{ok}\ \Theta \wedge x \notin \mathtt{dom}(\Theta)$.
By induction on the size of Θ:

- $\Theta = \emptyset$.
  $\Gamma \ominus x = \emptyset$. Immediate.
- $\Theta = (\Delta, z \mapsto t)$.
  $\Gamma \ominus x = ((\Delta, x \mapsto \mathtt{fvar}\ y) \ominus x, z \mapsto t[y/x])$.
  $\mathtt{ok}\ \Theta \Rightarrow \mathtt{ok}\ \Delta \wedge z \notin \mathtt{dom}(\Delta) \wedge \mathtt{lc}\ t$.
  $z \notin \mathtt{dom}(\Delta) \overset{L29}{=} \mathtt{dom}((\Delta, x \mapsto \mathtt{fvar}\ y) \ominus x)$.
  $\mathtt{lc}\ t \Rightarrow \mathtt{lc}\ t[y/x]$.
  $$\left.\begin{array}{l} \mathtt{ok}\ \Delta \\ x \notin \mathtt{dom}(\Theta) \Rightarrow x \notin \mathtt{dom}(\Delta) \\ \mathtt{lc}\ (\mathtt{fvar}\ y) \end{array}\right\} \Rightarrow \mathtt{ok}\ (\Delta, x \mapsto \mathtt{fvar}\ y) \overset{I.H.}{\Rightarrow} \mathtt{ok}\ ((\Delta, x \mapsto \mathtt{fvar}\ y) \ominus x).$$
  Therefore $\mathtt{ok}\ ((\Delta, x \mapsto \mathtt{fvar}\ y)) \ominus x, z \mapsto t[y/x]) \Rightarrow \mathtt{ok}\ (\Gamma \ominus x)$.

□

Lemma 31.
IND_SUBS $\quad x \notin \mathtt{dom}(\Gamma) \Rightarrow (\Gamma, x \mapsto \mathtt{fvar}\ y) \ominus x = \Gamma[y/x]$

Proof.
An easy induction on the size of Γ. □

Lemma 18
HCE_SWAP $\quad \mathtt{ok}\ \Gamma \wedge x, y \in \mathtt{Ind}(\Gamma) \wedge x \neq y \Rightarrow \Gamma \ominus [x, y] \approx^{V - \{x,y\}} \Gamma \ominus [y, x]$

Proof.
We have $\Gamma = (\Theta, x \mapsto \mathtt{fvar}\ x', y \mapsto \mathtt{fvar}\ y')$.
By induction on the size of Θ:

– $\Theta = \emptyset \Rightarrow \Gamma \ominus [x,y] = \emptyset = \Gamma \ominus [y,x]$
– $\Theta = (\Delta, z \mapsto t) \Rightarrow$
$\Gamma \ominus [x,y] = (\Gamma \ominus x) \ominus y$
$= ((\Delta, x \mapsto \texttt{fvar}\ x', y \mapsto \texttt{fvar}\ y') \ominus x, z \mapsto t[x'/x]) \ominus y$
$= (((\Delta, x \mapsto \texttt{fvar}\ x', y \mapsto \texttt{fvar}\ y') \ominus x) \ominus y, z \mapsto (t[x'/x])[y'[x'/x]/y])$
$= ((\Delta, x \mapsto \texttt{fvar}\ x', y \mapsto \texttt{fvar}\ y') \ominus [x,y], z \mapsto (t[x'/x])[y'[x'/x]/y])$
$= (\Delta' \ominus [x,y], z \mapsto t')$
where $\Delta' = (\Delta, x \mapsto \texttt{fvar}\ x', y \mapsto \texttt{fvar}\ y')$ and $t' = (t[x'/x])[y'[x'/x]/y]$.
Similarly, $\Gamma \ominus [y,x] = (\Delta' \ominus [y,x], z \mapsto t'')$ with $t'' = (t[y'/y])[x'[y'/y]/x]$.
$\texttt{ok}\ \Gamma \Rightarrow \texttt{ok}\ \Delta' \wedge z \notin \texttt{dom}(\Delta') \wedge \texttt{lc}\ t \overset{L29}{\Rightarrow} z \notin \texttt{dom}(\Delta' \ominus [x,y]) \wedge \texttt{lc}\ t'$.

$$\left.\begin{array}{l}\texttt{ok}\ \Delta' \\ x, y \in \texttt{Ind}(\Delta')\end{array}\right\} \overset{I.H.}{\Rightarrow} \Delta' \ominus [x,y] \approx^{V-\{x,y\}} \Delta' \ominus [y,x]$$

To prove: $t' \approx^{V-\{x,y\}} t''$.
- $x \neq y' \wedge y \neq x'$
$t' = (t[x'/x])[y'[x'/x]/y] = (t[x'/x])[y'/y] \overset{x \neq y}{=} (t[y'/y])[x'/x] = (t[y'/y])[x'[y'/y]/x] = t''$.
- $x \neq y' \wedge y = x'$
$t' = (t[x'/x])[y'[x'/x]/y] = (t[y/x])[y'/y] = (t[y'/x])[y'/y] \overset{x \neq y}{=} (t[y'/y])[y'/x] = (t[y'/y])[x'[y'/y]/x] = t''$.
- $x = y' \wedge y \neq x'$
$t' = (t[x'/x])[y'[x'/x]/y] = (t[x'/x])[x'/y] \overset{x \neq y}{=} (t[x'/y])[x'/x] = (t[x/y])[x'/x] = (t[x/y])[x'[x/y]/x] = (t[y'/y])[x'[y'/y]/x] = t''$.
- $x = y' \wedge y = x'$
$t' = (t[x'/x])[y'[x'/x]/y] = (t[y/x])[y/y] = t[y/x]$
$t'' = (t[y'/y])[x'[y'/y]/x] = (t[x/y])[x/x] = t[x/y]$
By CE_REF (Proposition 2),
$t \approx^{V-\{x,y\}} t \overset{L15}{\Rightarrow} t[y/x] \approx^{V-\{x,y\}} t \overset{P2}{\Rightarrow} t \approx^{V-\{x,y\}} t[y/x] \overset{L15}{\Rightarrow} t[x/y] \approx^{V-\{x,y\}} t[y/x]$.

$$\left.\begin{array}{l}\Delta' \ominus [x,y] \approx^{V-\{x,y\}} \Delta' \ominus [y,x] \\ t' \approx^{V-\{x,y\}} t'' \\ z \notin \texttt{dom}(\Delta' \ominus [x,y]) \wedge \texttt{lc}\ t'\end{array}\right\} \Rightarrow \Gamma \ominus [x,y] \approx^{V-\{x,y\}} \Gamma \ominus [y,x]$$

□

Lemma 19
HE_PERM $\quad \texttt{ok}\ \Gamma \wedge \overline{x}, \overline{y} \in \texttt{Ind}(\Gamma) \Rightarrow (\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \Leftrightarrow \overline{y} \in \mathcal{S}(\overline{x}))$

Proof.
$\Rightarrow$ $\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \overset{L16}{\Rightarrow} \texttt{dom}(\Gamma \ominus \overline{x}) = \texttt{dom}(\Gamma \ominus \overline{y}) \overset{L29}{\Rightarrow} \texttt{dom}(\Gamma) - \{\overline{x}\} = \texttt{dom}(\Gamma) - \{\overline{y}\}$.
Since $\overline{x}, \overline{y} \subseteq \texttt{dom}(\Gamma)$ and have pairwise-distinct names, we infer that $\overline{y} \in \mathcal{S}(\overline{x})$.

$\Leftarrow$ From Lemma 18 and Abstract Algebra results, i.e., any permutation can be written as a product of transpositions.

□

7.4 Results on indirection relation (Section 4.2)

Proposition 4
IR_ALT $\quad \Gamma \gtrsim_I \Gamma' \Leftrightarrow \texttt{ok}\ \Gamma \wedge \exists\ \overline{x} \subseteq \texttt{Ind}(\Gamma)\ .\ \Gamma \ominus \overline{x} \approx \Gamma'$

Proof.
$\Rightarrow$ By rule induction:

– IR_HE: $\Gamma \approx \Gamma'$.
  By HCE_OK (Lemma 16), $\mathtt{ok}\ \Gamma$.
  Let $\overline{x} = [\,]$, then $\Gamma \ominus [\,] = \Gamma \approx \Gamma'$.
– IR_IR: $\mathtt{ok}\ \Gamma \wedge \Gamma \ominus x \gtrsim_I \Gamma' \wedge x \in \mathtt{Ind}(\Gamma)$.
  By induction hypothesis,
  $\exists\ \overline{x} \subseteq \mathtt{Ind}(\Gamma \ominus x) \overset{L29}{=} \mathtt{Ind}(\Gamma) - \{x\}\ .\ (\Gamma \ominus x) \ominus \overline{x} \approx \Gamma' \Rightarrow \Gamma \ominus [x : \overline{x}] \approx \Gamma'$.

$\Leftarrow$ By induction on the length of $\overline{x}$:

– $\overline{x} = [\,]$
  $\Gamma = \Gamma \ominus [\,] \approx \Gamma' \Rightarrow \Gamma \gtrsim_I \Gamma'$.
– $\overline{x} = [y : \overline{y}]$

$$\left.\begin{array}{l} \mathtt{ok}\ \Gamma \wedge y \in \mathtt{Ind}(\Gamma) \overset{L30}{\Rightarrow} \mathtt{ok}\ (\Gamma \ominus y) \\ y \notin \overline{y} \wedge \overline{y} \subseteq \mathtt{Ind}(\Gamma) \Rightarrow \overline{y} \subseteq \mathtt{Ind}(\Gamma) - \{y\} \overset{L29}{=} \mathtt{Ind}(\Gamma \ominus y) \\ \Gamma \ominus [y : \overline{y}] = (\Gamma \ominus y) \ominus \overline{y} \end{array}\right\} \overset{I.H.}{\Rightarrow} (\Gamma \ominus y) \gtrsim_I \Gamma' \Rightarrow \Gamma \gtrsim_I \Gamma'.$$

□

Corollary 1.
IR_DOM_DOM $\quad \Gamma \gtrsim_I \Gamma' \Rightarrow \Gamma \ominus (\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')) \approx \Gamma'$

Proof.
$\Gamma \gtrsim_I \Gamma' \overset{P4}{\Rightarrow} \mathtt{ok}\ \Gamma \wedge \exists\ \overline{x} \subseteq \mathtt{Ind}(\Gamma)\ .\ \Gamma \ominus \overline{x} \approx \Gamma'$.
$\mathtt{dom}(\Gamma') \overset{L16}{=} \mathtt{dom}(\Gamma \ominus \overline{x}) \overset{L29}{=} \mathtt{dom}(\Gamma) - \overline{x} \Rightarrow \overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$. □

To prove Lemma 5 we extend some results on context-equivalence (for terms) to heap-context-equivalence.

Lemma 32.
HCE_SUB $\quad \Gamma \approx^V \Gamma' \wedge V' \subseteq V \Rightarrow \Gamma \approx^{V'} \Gamma'$
HCE_ADD $\quad \Gamma \approx^V \Gamma' \wedge \overline{x} \notin \mathtt{names}(\Gamma) \cup \mathtt{names}(\Gamma') \Rightarrow \Gamma \approx^{V \cup \overline{x}} \Gamma'$

Proof.

HCE_SUB
An easy induction using CE_SUB in Lemma 14.

HCE_ADD
An easy induction using CE_ADD in Lemma 14. □

Lemma 33.
HCE_DEL_IND $\quad \Gamma \approx^V \Gamma' \wedge \mathtt{dom}(\Gamma) \subseteq V \wedge x \in \mathtt{Ind}(\Gamma) \Rightarrow \Gamma \ominus x \approx^V \Gamma' \ominus x$
HE_DEL_IND $\quad \Gamma \approx \Gamma' \wedge x \in \mathtt{Ind}(\Gamma) \Rightarrow \Gamma \ominus x \approx \Gamma' \ominus x$

Proof.
HCE_DEL_IND
$x \in \mathtt{Ind}(\Gamma) \overset{L16}{=} \mathtt{Ind}(\Gamma') \Rightarrow \Gamma = (\Theta, x \mapsto \mathtt{fvar}\ y) \wedge \Gamma' = (\Theta', x \mapsto \mathtt{fvar}\ y')$
$\overset{L28}{\Rightarrow} \Theta \approx^V \Theta' \wedge (\mathtt{fvar}\ y) \approx^V (\mathtt{fvar}\ y')$.
Moreover, by HCE_OK (Lemma 16) $x \notin \mathtt{dom}(\Theta) \cup \mathtt{dom}(\Theta')$.
In addition, $\mathtt{dom}(\Gamma) \subseteq V \Rightarrow x \in V$.
We proceed by induction on the size of Θ:

– $\Theta = \emptyset \overset{L16}{\Rightarrow} \Theta' = \emptyset$.

– $\Theta = (\Delta, z \mapsto t)$ with $z \neq x$.
  By Lemma 16, $\Theta' = (\Delta', z \mapsto t')$.
  By Lemma 28, $\Delta \approx^V \Delta' \wedge t \approx^V t'$.
  Therefore, $(\Delta, x \mapsto \texttt{fvar}\ y) \approx^V (\Delta', x \mapsto \texttt{fvar}\ y') \overset{I.H.}{\Rightarrow} (\Delta, x \mapsto \texttt{fvar}\ y) \ominus x \approx^V (\Delta', x \mapsto \texttt{fvar}\ y') \ominus x$.
  Moreover, $t \approx^V t' \wedge (\texttt{fvar}\ y) \approx^V (\texttt{fvar}\ y') \wedge x \in V \overset{L15}{\Rightarrow} t[y/x] \approx^V t'[y'/x]$, $\texttt{lc}\ t \Rightarrow \texttt{lc}\ t[y/x]$, and $z \notin \texttt{dom}(\Delta, x \mapsto \texttt{fvar}\ y)$.
  Hence, $((\Delta, x \mapsto \texttt{fvar}\ y) \ominus x, z \mapsto t[y/x]) \approx^V ((\Delta', x \mapsto \texttt{fvar}\ y') \ominus x, z \mapsto t'[y'/x])$.
  But $\Gamma \ominus x = ((\Delta, x \mapsto \texttt{fvar}\ y) \ominus x, z \mapsto t[y/x])$ and $\Gamma' \ominus x = ((\Delta', x \mapsto \texttt{fvar}\ y') \ominus x, z \mapsto t'[y'/x])$.

HE_DEL_IND
It is a direct consequence of HCE_DEL_IND and Lemma 32. □

Proposition 5.

IR_REF	$\texttt{ok}\ \Gamma \Rightarrow \Gamma \gtrsim_I \Gamma$
IR_TRANS	$\Gamma \gtrsim_I \Gamma' \wedge \Gamma' \gtrsim_I \Gamma'' \Rightarrow \Gamma \gtrsim_I \Gamma''$

Proof.
IR_REF
By HCE_REF in Proposition 3.

IR_TRANS
By rule induction on $\Gamma \gtrsim_I \Gamma'$:

– $\Gamma \approx \Gamma'$.
  $\Gamma' \gtrsim_I \Gamma'' \overset{P4}{\Rightarrow} \texttt{ok}\ \Gamma' \wedge \exists\ \overline{x} \subseteq \texttt{Ind}(\Gamma')\ .\ \Gamma' \ominus \overline{x} \approx \Gamma''$.
  By HCE_IND (Lemma 16), $\overline{x} \subseteq \texttt{Ind}(\Gamma)$.
  By HE_DEL_IND (Lemma 33), $\Gamma \ominus \overline{x} \approx \Gamma' \ominus \overline{x}$ and, by transitivity of $\approx$ (Proposition 3), $\Gamma \ominus \overline{x} \approx \Gamma''$.
  By Proposition 4 (in the other direction) we get $\Gamma \gtrsim_I \Gamma''$.
– $\Gamma \ominus y \gtrsim_I \Gamma' \wedge y \in \texttt{Ind}(\Gamma) \wedge \texttt{ok}\ \Gamma$.
  By induction hypothesis $\Gamma \ominus y \gtrsim_I \Gamma'' \Rightarrow \Gamma \gtrsim_I \Gamma''$.

□

Lemma 20

IR_DOM	$\Gamma \gtrsim_I \Gamma' \Rightarrow \texttt{dom}(\Gamma') \subseteq \texttt{dom}(\Gamma)$
IR_IND	$\Gamma \gtrsim_I \Gamma' \Rightarrow \texttt{Ind}(\Gamma') \subseteq \texttt{Ind}(\Gamma)$
IR_OK	$\Gamma \gtrsim_I \Gamma' \Rightarrow \texttt{ok}\ \Gamma \wedge \texttt{ok}\ \Gamma'$
IR_DOM_HE	$\Gamma \gtrsim_I \Gamma' \wedge \texttt{dom}(\Gamma) = \texttt{dom}(\Gamma') \Rightarrow \Gamma \approx \Gamma'$
IR_IR_HE	$(\Gamma \gtrsim_I \Gamma' \wedge \Gamma' \gtrsim_I \Gamma) \Leftrightarrow \Gamma \approx \Gamma'$

Proof.

The proofs of IR_DOM, IR_IND and IR_OK are easy rule inductions.

IR_DOM_HE
We prove by contrapositive: Assume $\Gamma \gtrsim_I \Gamma' \wedge \Gamma \not\approx \Gamma'$.
$\Gamma \gtrsim_I \Gamma' \Rightarrow \texttt{ok}\ \Gamma \wedge \Gamma \ominus x \gtrsim_I \Gamma' \wedge x \in \texttt{Ind}(\Gamma)$.
By IR_DOM, $\texttt{dom}(\Gamma') \subseteq \texttt{dom}(\Gamma \ominus x) \overset{L29}{=} \texttt{dom}(\Gamma) - \{x\} \subsetneq \texttt{dom}(\Gamma)$.
Hence $\texttt{dom}(\Gamma) \neq \texttt{dom}(\Gamma')$.

IR_IR_HE

$\Rightarrow$ By IR_DOM,
$\left.\begin{array}{l} \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{dom}(\Gamma') \subseteq \mathtt{dom}(\Gamma) \\ \Gamma' \succsim_I \Gamma \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Gamma') \end{array}\right\} \Rightarrow \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma')$.
By IR_DOM_HE, $\Gamma \approx \Gamma'$.

$\Leftarrow$ $\Gamma \approx \Gamma' \Rightarrow \Gamma \succsim_I \Gamma'$.
$\Gamma \approx \Gamma' \overset{P3}{\Rightarrow} \Gamma' \approx \Gamma \Rightarrow \Gamma' \succsim_I \Gamma$.

□

Lemma 21

IREQ_HE_IREQ1	$[\Gamma] \succsim_I [\Gamma'] \wedge \Delta \approx \Gamma \Rightarrow [\Delta] \succsim_I [\Gamma']$
IREQ_HE_IREQ2	$[\Gamma] \succsim_I [\Gamma'] \wedge \Delta \approx \Gamma' \Rightarrow [\Gamma] \succsim_I [\Delta]$

Proof.
IREQ_HE_IREQ1
$[\Gamma] \succsim_I [\Gamma'] \Rightarrow \Gamma \succsim_I \Gamma'$.

- $\Gamma \approx \Gamma'$.
  $\Delta \approx \Gamma \overset{L16}{\Rightarrow} \mathtt{dom}(\Delta) = \mathtt{dom}(\Gamma) \overset{P3}{\Rightarrow} \Delta \approx \Gamma' \Rightarrow \Delta \succsim_I \Gamma' \Rightarrow [\Delta] \succsim_I [\Gamma']$.
- $\mathtt{ok}\ \Gamma \wedge (\Gamma \ominus x) \succsim_I \Gamma' \wedge x \in \mathtt{Ind}(\Gamma)$.
  $\Delta \approx \Gamma \overset{L16}{\Rightarrow} x \in \mathtt{Ind}(\Delta) \wedge \mathtt{ok}\ \Delta \overset{L33}{\Rightarrow} (\Delta \ominus x) \approx (\Gamma \ominus x) \Rightarrow (\Delta \ominus x) \succsim_I (\Gamma \ominus x) \overset{P5}{\Rightarrow} (\Delta \ominus x) \succsim_I \Gamma'$.
  Thus, $\Delta \succsim_I \Gamma' \Rightarrow [\Delta] \succsim_I [\Gamma']$.

IREQ_HE_IREQ2
$[\Gamma] \succsim_I [\Gamma'] \Rightarrow \Gamma \succsim_I \Gamma'$.

- $\Gamma \approx \Gamma' \overset{L16}{\Rightarrow} \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma')$.
  $\Delta \approx \Gamma' \overset{P3}{\Rightarrow} \Gamma' \approx \Delta \overset{P3}{\Rightarrow} \Gamma \approx \Delta \Rightarrow \Gamma \succsim_I \Delta \Rightarrow [\Gamma] \succsim_I [\Delta]$.
- $\mathtt{ok}\ \Gamma \wedge (\Gamma \ominus x) \succsim_I \Gamma' \wedge x \in \mathtt{Ind}(\Gamma)$.
  $\Delta \approx \Gamma' \overset{P3}{\Rightarrow} \Gamma' \approx \Delta \Rightarrow \Gamma' \succsim_I \Delta \overset{P5}{\Rightarrow} (\Gamma \ominus x) \succsim_I \Delta \Rightarrow \Gamma \succsim_I \Delta \Rightarrow [\Gamma] \succsim_I [\Delta]\,\cdot$

□

Proposition 6

IREQ_REF	$\mathtt{ok}\ \Gamma \Rightarrow [\Gamma] \succsim_I [\Gamma]$
IREQ_ANTSYM	$[\Gamma] \succsim_I [\Gamma'] \wedge [\Gamma'] \succsim_I [\Gamma] \Rightarrow [\Gamma] = [\Gamma']$
IREQ_TRANS	$[\Gamma] \succsim_I [\Gamma'] \wedge [\Gamma'] \succsim_I [\Gamma''] \Rightarrow [\Gamma] \succsim_I [\Gamma'']$

Proof.
Reflexivity and transitivity are immediate because $\succsim_I$ is a preorder for heaps (Proposition 5).
Antisymmetry is a consequence of IR_IR_HE (Lemma 20). □

Lemma 22

IRHT_IRH	$(\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow \Gamma \succsim_I \Gamma'$
IRHT_SS	$(\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow t \sim_S t'$
IRHT_LC	$(\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow \mathtt{lc}\ t \wedge \mathtt{lc}\ t'$

Proof.
$(\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow \forall z \notin L\,.\,(\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t')$, for some finite $L \subseteq Id$.

IRHT_IRH
$(\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t') \overset{C1}{\Rightarrow} (\Gamma, z \mapsto t) \ominus \overline{x} \approx (\Gamma', z \mapsto t')$ with $\overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$.
$(\Gamma, z \mapsto t) \ominus \overline{x} = (\Gamma \ominus \overline{x}, z \mapsto t'')$ being t'' the transformation of t by $\overline{x}$
$\overset{L28}{\Rightarrow} \Gamma \ominus \overline{x} \approx \Gamma' \overset{P4}{\Rightarrow} \Gamma \succsim_I \Gamma'$.

IRHT_SS
$(\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t') \overset{C1}{\Rightarrow} (\Gamma, z \mapsto t) \ominus \overline{x} \approx (\Gamma', z \mapsto t')$ with $\overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$.
We proceed by induction on the length of $\overline{x}$:

- $\overline{x} = [\,] \Rightarrow (\Gamma, z \mapsto t) \approx (\Gamma', z \mapsto t') \overset{L28}{\Rightarrow} t \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} t' \overset{L13}{\Rightarrow} t \sim_S t'$.
- $\overline{x} = [y : \overline{y}]$ for some $y \mapsto \mathtt{fvar}\ y' \in \Gamma$.
  $$\begin{aligned}(\Gamma, z \mapsto t) \ominus \overline{x} &= ((\Gamma, z \mapsto t) \ominus y) \ominus \overline{y} \\ &= (\Gamma \ominus y, z \mapsto t[y'/y]) \ominus \overline{y} \approx (\Gamma', z \mapsto t') \\ &\overset{I.H.}{\Rightarrow} t[y'/y] \sim_S t' \overset{L1\&P1}{\Rightarrow} t \sim_S t'.\end{aligned}$$

IRHT_LC
$(\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t') \overset{L20}{\Rightarrow} \mathtt{ok}\ (\Gamma, z \mapsto t) \wedge \mathtt{ok}\ (\Gamma', z \mapsto t') \Rightarrow \mathtt{lc}\ t \wedge \mathtt{lc}\ t'$. □

7.5 Equivalence (Section 4.3)

Several auxiliary results are needed to prove Proposition 7.

If $(\Gamma : \mathtt{fvar}\ x)$ and $(\Gamma' : \mathtt{fvar}\ x')$ are related, and there is a binding for x' in Γ', then there must be a binding for x in Γ too. Furthermore, there exists in Γ a sequence of indirections leading from x to x'.

Lemma 34.
IR_FVAR
$$\begin{aligned}&(\Gamma : \mathtt{fvar}\ x) \succsim_I (\Gamma' : \mathtt{fvar}\ x') \\ &\Rightarrow (x \notin \mathtt{dom}(\Gamma) \wedge x' \notin \mathtt{dom}(\Gamma')) \\ &\vee (x \in \mathtt{dom}(\Gamma) \wedge x' \in \mathtt{dom}(\Gamma') \wedge \\ &\quad \exists\, x_0, \ldots, x_n \in Id\,.\, x_0 = x \wedge x_n = x' \wedge \forall i : 0 \le i < n\,.\, x_i \mapsto \mathtt{fvar}\ x_{i+1} \in \Gamma)\end{aligned}$$

Proof.
$(\Gamma : \mathtt{fvar}\ x) \succsim_I (\Gamma' : \mathtt{fvar}\ x') \Rightarrow \forall z \notin L\,.\, (\Gamma, z \mapsto \mathtt{fvar}\ x) \succsim_I (\Gamma', z \mapsto \mathtt{fvar}\ x')$
$\overset{C1}{\Rightarrow} (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$ with $\overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$.
Take $z \notin L$ such that $z \neq x \wedge z \neq x'$. Now we proceed by induction on the length of $\overline{x}$:

- $\overline{x} = [\,] \Rightarrow \mathtt{dom}(\Gamma') = \mathtt{dom}(\Gamma) \wedge (\Gamma, z \mapsto \mathtt{fvar}\ x) \approx (\Gamma', z \mapsto \mathtt{fvar}\ x') \overset{L28}{\Rightarrow} \mathtt{fvar}\ x \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} \mathtt{fvar}\ x'$.
  - $x \notin \mathtt{dom}(\Gamma) \Rightarrow x' \notin \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma')$.
  - $x \in \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \Rightarrow x = x' \Rightarrow x' \in \mathtt{dom}(\Gamma')$,
    and the second part of the result is immediate (take $n = 0$).
- $\overline{x} \neq [\,]$
  - $x \notin \overline{x} \Rightarrow (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x} = (\Gamma \ominus \overline{x}, z \mapsto \mathtt{fvar}\ x) \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$.
    With a reasoning similar to the empty list case, we infer that either $x \notin \mathtt{dom}(\Gamma) \wedge x' \notin \mathtt{dom}(\Gamma')$, or $x \in \mathtt{dom}(\Gamma) \cap \mathtt{dom}(\Gamma')$ and $x = x'$.
  - $x \in \overline{x} \Rightarrow x \in \mathtt{Ind}(\Gamma)$ and $x \mapsto \mathtt{fvar}\ y \in \Gamma$ for some $y \in Id$.
    Moreover, we have $[x : \overline{y}] \in \mathcal{S}(\overline{x}) \overset{L19}{\Rightarrow} (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus [x : \overline{y}] \approx (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x}$.
    Hence $(\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus [x : \overline{y}] = (\Gamma \ominus x, z \mapsto \mathtt{fvar}\ y) \ominus \overline{y}$
    $\overset{I.H.}{\Rightarrow} x' \in \mathtt{dom}(\Gamma') \wedge \exists\, x_0, \ldots, x_n \in Id\,.\, x_0 = y \wedge x_n = x' \wedge \forall i : 0 \le i < n\,.\, x_i \mapsto \mathtt{fvar}\ x_{i+1} \in \Gamma \ominus x$.
    Since $\forall i : 0 < i < n\,.\, x_i \neq y$ we have $\forall i : 0 \le i < n\,.\, x_i \mapsto \mathtt{fvar}\ x_{i+1} \in \Gamma$ too, and we can extend the sequence with $x \mapsto \mathtt{fvar}\ y$.

□

Lemma 35.
IR_IRHT $(\Gamma, x \mapsto t) \succsim_I (\Gamma', x \mapsto t') \Rightarrow ((\Gamma, x \mapsto t) : t) \succsim_I ((\Gamma', x \mapsto t') : t')$

Proof. By rule induction:

– IR_HE
$(\Gamma, x \mapsto t) \approx (\Gamma', x \mapsto t') \Rightarrow (\Gamma, x \mapsto t) \approx^{\mathtt{dom}(\Gamma, x\mapsto t)} (\Gamma', x \mapsto t') \overset{L28}{\Rightarrow} t \approx^{\mathtt{dom}(\Gamma, x\mapsto t)} t' \wedge \mathtt{lc}\ t$.
Let $L = \mathtt{names}(\Gamma, x \mapsto t) \cup \mathtt{names}(\Gamma', x \mapsto t')$, then by CE_ADD (in Lemma 14) and HCE_ADD (in Lemma 32), $\forall z \notin L\,.\,(\Gamma, x \mapsto t) \approx^{\mathtt{dom}(\Gamma, x\mapsto t)\cup\{z\}} (\Gamma', x \mapsto t') \wedge t \approx^{\mathtt{dom}(\Gamma, x\mapsto t)\cup\{z\}} t'$
$\Rightarrow \forall z \notin L\,.\,(\Gamma, x \mapsto t, z \mapsto t) \approx^{\mathtt{dom}(\Gamma, x\mapsto t)\cup\{z\}} (\Gamma', x \mapsto t', z \mapsto t')$
$\Rightarrow \forall z \notin L\,.\,(\Gamma, x \mapsto t, z \mapsto t) \gtrsim_I (\Gamma', x \mapsto t', z \mapsto t')$
$\Rightarrow ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x \mapsto t') : t')$.
– IR_IR
$(\Gamma, x \mapsto t) \ominus y \gtrsim_I (\Gamma', x \mapsto t')$ for some $y \mapsto \mathtt{fvar}\ y' \in \Gamma$
$\Rightarrow (\Gamma \ominus y, x \mapsto t[y'/y]) \gtrsim_I (\Gamma', x \mapsto t')$
$\overset{I.H.}{\Rightarrow} ((\Gamma \ominus y, x \mapsto t[y'/y]) : t[y'/y]) \gtrsim_I ((\Gamma', x \mapsto t') : t')$
$\Rightarrow \forall z \notin L\,.\,(\Gamma \ominus y, x \mapsto t[y'/y], z \mapsto t[y'/y]) \gtrsim_I (\Gamma', x \mapsto t', z \mapsto t')$, for some $L \subseteq Id$
$\Rightarrow \forall z \notin L \cup \mathtt{dom}(\Gamma)\,.\,(\Gamma, x \mapsto t, z \mapsto t) \ominus y \gtrsim_I (\Gamma', x \mapsto t', z \mapsto t')$
$\Rightarrow ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x \mapsto t') : t')$.

□

Lemma 36.
IR_FVAR_IRHT $((\Gamma, x \mapsto t) : \mathtt{fvar}\ x) \gtrsim_I ((\Gamma', x' \mapsto t') : \mathtt{fvar}\ x')$
$\Rightarrow ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x' \mapsto t') : t') \vee ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x' \mapsto t') : \mathtt{fvar}\ x')$.

Proof.
$((\Gamma, x \mapsto t) : \mathtt{fvar}\ x) \gtrsim_I ((\Gamma', x' \mapsto t') : \mathtt{fvar}\ x')$
$\overset{L34}{\Rightarrow} \exists x_0, \ldots, x_n \in Id\,.\,x_0 = x \wedge x_n = x' \wedge \forall i : 0 \leq i < n\,.\,x_i \mapsto \mathtt{fvar}\ x_{i+1} \in (\Gamma, x \mapsto t)$.

– $n = 0 \Rightarrow x = x'$
$\Rightarrow ((\Gamma, x \mapsto t) : \mathtt{fvar}\ x) \gtrsim_I ((\Gamma', x \mapsto t') : \mathtt{fvar}\ x)$
$\overset{L22}{\Rightarrow} (\Gamma, x \mapsto t) \gtrsim_I (\Gamma', x \mapsto t')$
$\overset{L35}{\Rightarrow} ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x \mapsto t') : t')$.
– $n > 0 \Rightarrow t \equiv \mathtt{fvar}\ x_1 \Rightarrow ((\Gamma, x \mapsto \mathtt{fvar}\ x_1) : \mathtt{fvar}\ x) \gtrsim_I ((\Gamma', x' \mapsto t') : \mathtt{fvar}\ x')$
$\Rightarrow \forall z \notin L\,.\,(\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x) \gtrsim_I (\Gamma', x' \mapsto t', z \mapsto \mathtt{fvar}\ x')$ for some $L \subseteq Id$
$\overset{C1}{\Rightarrow} (\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x) \ominus \overline{y} \approx (\Gamma', x' \mapsto t', z \mapsto \mathtt{fvar}\ x')$
with $\overline{y} = \mathtt{dom}(\Gamma) \cup \{x\} - (\mathtt{dom}(\Gamma') \cup \{x'\})$.
  • $x \notin \overline{y} \Rightarrow x \in \mathtt{dom}((\Gamma, x \mapsto \mathtt{fvar}\ x_1) \ominus \overline{y})$.
  $(\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x) \ominus \overline{y} = ((\Gamma, x \mapsto \mathtt{fvar}\ x_1) \ominus \overline{y}, z \mapsto \mathtt{fvar}\ x)$
  $\overset{L28}{\Rightarrow} \mathtt{fvar}\ x \approx^{\mathtt{dom}((\Gamma, x\mapsto \mathtt{fvar}\ x_1)\ominus\overline{y})\cup\{z\}} \mathtt{fvar}\ x'$.
  Hence, $x = x'$ and we proceed like in the case $n = 0$.
  • $x \in \overline{y} \Rightarrow [x : \overline{x}] \in \mathcal{S}(\overline{y})$ for some $\overline{x}$.
  $\overset{L19}{\Rightarrow} (\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x) \ominus [x : \overline{x}] \approx (\Gamma', x' \mapsto t', z \mapsto \mathtt{fvar}\ x')$.
  But $(\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x) \ominus [x : \overline{x}] = (\Gamma, z \mapsto \mathtt{fvar}\ x_1) \ominus \overline{x} =$
  $(\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x_1) \ominus [x : \overline{x}]$
  $\overset{P4}{\Rightarrow} (\Gamma, x \mapsto \mathtt{fvar}\ x_1, z \mapsto \mathtt{fvar}\ x_1) \gtrsim_I (\Gamma', x' \mapsto t', z \mapsto \mathtt{fvar}\ x')$.
  This can be obtained for any $z \notin L$, so that
  $((\Gamma, x \mapsto \mathtt{fvar}\ x_1) : \mathtt{fvar}\ x_1) \gtrsim_I ((\Gamma', x' \mapsto t') : \mathtt{fvar}\ x')$.

□

If two heap/application pairs are indirection related then the bodies of the applications and their arguments are also related.

Lemma 37.
IRHT_APP $(\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \gtrsim_I (\Gamma' : \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$
$\Rightarrow (\Gamma : t) \gtrsim_I (\Gamma' : t') \wedge (\Gamma : \mathtt{fvar}\ x) \gtrsim_I (\Gamma' : \mathtt{fvar}\ x')$.

Proof.
$(\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \precsim_I (\Gamma' : \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$
$\Rightarrow \forall z \notin L\,.\,(\Gamma, z \mapsto \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \precsim_I (\Gamma', z \mapsto \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$ for some finite $L \subseteq Id$
$\overset{C1}{\Rightarrow} (\Gamma, z \mapsto \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$ with $\overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$.
Now we prove by induction on the length of $\overline{x}$ that

$$(\Gamma, z \mapsto \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$$
$$\Rightarrow (\Gamma, z \mapsto t) \ominus \overline{x} \approx (\Gamma', z \mapsto t') \wedge (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$$

- $\overline{x} = [\,]$
  $\Rightarrow (\Gamma, z \mapsto \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \approx (\Gamma', z \mapsto \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$
  $\overset{L28}{\Rightarrow} \Gamma \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} \Gamma' \wedge (\mathtt{app}\ t\ (\mathtt{fvar}\ x)) \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} (\mathtt{app}\ t'\ (\mathtt{fvar}\ x')) \wedge \mathtt{lc}\ (\mathtt{app}\ t\ (\mathtt{fvar}\ x)) \wedge z \notin \mathtt{dom}(\Gamma)$
  $(\mathtt{app}\ t\ (\mathtt{fvar}\ x)) \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} (\mathtt{app}\ t'\ (\mathtt{fvar}\ x')) \Rightarrow t \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} t' \wedge (\mathtt{fvar}\ x) \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} (\mathtt{fvar}\ x')$
  $\mathtt{lc}\ (\mathtt{app}\ t\ (\mathtt{fvar}\ x)) \Rightarrow \mathtt{lc}\ t \wedge \mathtt{lc}\ (\mathtt{fvar}\ x)$
  $\Rightarrow (\Gamma, z \mapsto t) \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} (\Gamma', z \mapsto t') \wedge (\Gamma, z \mapsto \mathtt{fvar}\ x) \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} (\Gamma', z \mapsto \mathtt{fvar}\ x')$.
- $\overline{x} = [y : \overline{y}]$ with $y \mapsto \mathtt{fvar}\ y' \in \Gamma \wedge y \neq z$.
  $(\Gamma, z \mapsto \mathtt{app}\ t\ (\mathtt{fvar}\ x)) \ominus \overline{x} = (\Gamma \ominus y, z \mapsto \mathtt{app}\ t[y'/y]\ (\mathtt{fvar}\ x[y'/y])) \ominus \overline{y} \approx (\Gamma', z \mapsto \mathtt{app}\ t'\ (\mathtt{fvar}\ x'))$
  $\overset{I.H.}{\Rightarrow} (\Gamma \ominus y, z \mapsto t[y'/y]) \ominus \overline{y} \approx (\Gamma', z \mapsto t') \wedge (\Gamma \ominus y, z \mapsto \mathtt{fvar}\ x[y'/y]) \ominus \overline{y} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$
  $\Rightarrow (\Gamma, z \mapsto t) \ominus [y : \overline{y}] \approx (\Gamma', z \mapsto t') \wedge (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus [y : \overline{y}] \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$.

By Proposition 4 we have $(\Gamma, z \mapsto t) \precsim_I (\Gamma', z \mapsto t') \wedge (\Gamma, z \mapsto \mathtt{fvar}\ x) \precsim_I (\Gamma', z \mapsto \mathtt{fvar}\ x')$,
and the result is valid for any $z \notin L$, so that $(\Gamma : t) \precsim_I (\Gamma' : t')$ and $(\Gamma : \mathtt{fvar}\ x) \precsim_I (\Gamma' : \mathtt{fvar}\ x')$. □

The next lemma is useful to show that the indirection relation between variables can be preserved through evaluation.

Lemma 38.
IRHT_FV
$$(\Gamma : \mathtt{fvar}\ x) \precsim_I (\Gamma' : \mathtt{fvar}\ x') \wedge \Gamma \subseteq \Delta \wedge \Gamma' \subseteq \Delta' \wedge \Delta \precsim_I \Delta' \wedge$$
$$(x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin \mathtt{dom}(\Delta)) \wedge (x' \notin \mathtt{dom}(\Gamma') \Rightarrow x' \notin \mathtt{dom}(\Delta')) \wedge$$
$$(y \in \mathtt{dom}(\Gamma) \wedge y \notin \mathtt{dom}(\Gamma') \Rightarrow y \notin \mathtt{dom}(\Delta'))$$
$$\Rightarrow (\Delta : \mathtt{fvar}\ x) \precsim_I (\Delta' : \mathtt{fvar}\ x').$$

Proof.
$(\Gamma : \mathtt{fvar}\ x) \precsim_I (\Gamma' : \mathtt{fvar}\ x') \Rightarrow \forall z \notin L\,.\,(\Gamma, z \mapsto \mathtt{fvar}\ x) \precsim_I (\Gamma', z \mapsto \mathtt{fvar}\ x')$ for some finite $L \subseteq Id$
$\overset{C1}{\Rightarrow} \forall z \notin L\,.\,(\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$ with $\overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$.
Furthermore, $\Delta \precsim_I \Delta' \overset{C1}{\Rightarrow} \Delta \ominus \overline{y} \approx \Delta'$ with $\overline{y} = \mathtt{dom}(\Delta) - \mathtt{dom}(\Delta')$.
Notice that, $\Gamma \subseteq \Delta \Rightarrow \overline{x} \subseteq \mathtt{dom}(\Delta)$;
and by hypothesis, $y \in \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma') \Rightarrow y \notin \mathtt{dom}(\Delta')$, so that we can write $\overline{y} = \overline{x} +\!\!+ \overline{z}$.
Let $L' = L \cup \{x, x'\} \cup \mathtt{dom}(\Delta) \cup \overline{y}$, we prove:

$$\forall z \notin L'\,.\,(\Delta, z \mapsto \mathtt{fvar}\ x) \ominus \overline{y} \approx (\Delta', z \mapsto \mathtt{fvar}\ x').$$

By induction on the length of $\overline{x}$:

- $\overline{x} = [\,] \Rightarrow \overline{y} = \overline{z}$.
  On the one hand, $z \notin L' \Rightarrow z \notin L$ so that $(\Gamma, z \mapsto \mathtt{fvar}\ x) \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$
  $\overset{L28}{\Rightarrow} \mathtt{fvar}\ x \approx^{\mathtt{dom}(\Gamma)\cup\{z\}} \mathtt{fvar}\ x'$
  $\Rightarrow x = x' \vee x, x' \notin \mathtt{dom}(\Gamma) \cup \{z\} = \mathtt{dom}(\Gamma') \cup \{z\}$.
  On the other hand, $\Delta \ominus \overline{z} \approx \Delta'$.
  We show that $\mathtt{fvar}\ x \approx^{\mathtt{dom}(\Delta\ominus\overline{z})\cup\{z\}} \mathtt{fvar}\ x'$:
  - If $x = x'$ then the result is trivial.

- If $x, x' \notin \mathtt{dom}(\Gamma) \cup \{z\} = \mathtt{dom}(\Gamma') \cup \{z\}$ then, by hypothesis,
  $x \notin \mathtt{dom}(\Delta) \cup \{z\} \Rightarrow x \notin \mathtt{dom}(\Delta \ominus \overline{z}) \cup \{z\}$
  $x' \notin \mathtt{dom}(\Delta') \cup \{z\} = \mathtt{dom}(\Delta \ominus \overline{z}) \cup \{z\}$

We know that $\mathtt{lc}\ (\mathtt{fvar}\ x)$ and $z \notin L' \Rightarrow z \notin \mathtt{dom}(\Delta \ominus \overline{z})$.
Therefore, $\forall z \notin L'.(\Delta, z \mapsto \mathtt{fvar}\ x) \ominus \overline{z} = (\Delta \ominus \overline{z}, z \mapsto \mathtt{fvar}\ x) \approx (\Delta', z \mapsto \mathtt{fvar}\ x')$.

– $\overline{x} = [y : \overline{x'}]$ with $y \mapsto \mathtt{fvar}\ y' \in \Gamma$.
We know that:
1. $(\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus [y : \overline{x'}] = (\Gamma \ominus y, z \mapsto \mathtt{fvar}\ x[y'/y]) \ominus \overline{x'} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$, for all $z \notin L' \subseteq L$.
2. $\Gamma \subseteq \Delta \Rightarrow \Gamma \ominus y \subseteq \Delta \ominus y$.
3. $\Delta \ominus ([y : \overline{x'}] +\!\!+ \overline{z}) = (\Delta \ominus y) \ominus (\overline{x'} +\!\!+ \overline{z}) \approx \Delta' \overset{P4}{\Rightarrow} \Delta \ominus y \gtrsim_I \Delta'$.
4. By Lemma 29, $\mathtt{dom}(\Gamma \ominus y) = \mathtt{dom}(\Gamma) - \{y\} \wedge \mathtt{dom}(\Delta \ominus y) = \mathtt{dom}(\Delta) - \{y\}$.
   Therefore, $x \notin \mathtt{dom}(\Gamma \ominus y) \Rightarrow x \notin \mathtt{dom}(\Gamma) \vee x = y \Rightarrow x \notin \mathtt{dom}(\Delta) \vee x = y \Rightarrow x \notin \mathtt{dom}(\Delta \ominus y)$.
5. $y'' \in \mathtt{dom}(\Gamma \ominus y) \wedge y'' \notin \mathtt{dom}(\Gamma') \Rightarrow y'' \in \mathtt{dom}(\Gamma) \wedge y'' \neq y \wedge y'' \notin \mathtt{dom}(\Gamma') \Rightarrow y'' \notin \mathtt{dom}(\Delta')$.

Then by induction hypothesis:
$\forall z \notin L'.(\Delta \ominus y, z \mapsto (\mathtt{fvar}\ x)[y'/y]) \ominus (\overline{x'} +\!\!+ \overline{z}) \approx (\Delta', z \mapsto \mathtt{fvar}\ x')$
$\Rightarrow \forall z \notin L'.(\Delta, z \mapsto \mathtt{fvar}\ x) \ominus \overline{y} \approx (\Delta', z \mapsto \mathtt{fvar}\ x')$.

Now, by Proposition 4, $\forall z \notin L'.(\Delta, z \mapsto \mathtt{fvar}\ x) \gtrsim_I (\Delta', z \mapsto \mathtt{fvar}\ x') \Rightarrow (\Delta : \mathtt{fvar}\ x) \gtrsim_I (\Delta' : \mathtt{fvar}\ x')$. □

The introduction of indirections at functional application preserves the indirection-relation.

Lemma 39.
IRHT_RED_IND
$\mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge y \neq x \wedge (\Gamma : \mathtt{abs}\ t) \gtrsim_I (\Gamma' : \mathtt{abs}\ t') \wedge (\Gamma : \mathtt{fvar}\ x) \gtrsim_I (\Gamma' : \mathtt{fvar}\ x')$
$\Rightarrow ((\Gamma, y \mapsto \mathtt{fvar}\ x) : t^y) \gtrsim_I (\Gamma' : t'^{x'})$.

Proof.
$(\Gamma : \mathtt{abs}\ t) \gtrsim_I (\Gamma' : \mathtt{abs}\ t') \Rightarrow \forall z \notin L'.(\Gamma, z \mapsto \mathtt{abs}\ t) \gtrsim_I (\Gamma', z \mapsto \mathtt{abs}\ t')$, for some finite $L' \subseteq Id$
$\overset{C1}{\Rightarrow} \forall z \notin L'.(\Gamma, z \mapsto \mathtt{abs}\ t) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{abs}\ t')$ with $\overline{x} = \mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$.
Similarly, $(\Gamma : \mathtt{fvar}\ x) \gtrsim_I (\Gamma' : \mathtt{fvar}\ x') \Rightarrow \forall z \notin L''.(\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$ for some finite $L'' \subseteq Id$.
Let $L = L' \cup L'' \cup \{y\} \cup \mathtt{names}(\Gamma) \cup \mathtt{names}(\Gamma') \cup \mathtt{fv}(t) \cup \mathtt{fv}(t') \cup \{x, x'\}$,
we will prove that $\forall z \notin L.(\Gamma, y \mapsto \mathtt{fvar}\ x, z \mapsto t^y) \ominus [y : \overline{x}] \approx (\Gamma', z \mapsto t'^{x'})$.
By induction on the length of $\overline{x}$:

– $\overline{x} = [\,]$.
Let $z \notin L$, $(\Gamma, y \mapsto \mathtt{fvar}\ x, z \mapsto t^y) \ominus y = ((\Gamma, y \mapsto \mathtt{fvar}\ x) \ominus y, z \mapsto t^x) \overset{L31}{=} (\Gamma, z \mapsto t^x)$.
$z \notin L \Rightarrow z \notin L' \Rightarrow (\Gamma, z \mapsto \mathtt{abs}\ t) \approx (\Gamma', z \mapsto \mathtt{abs}\ t')$
$\overset{L28}{\Rightarrow} (\mathtt{abs}\ t) \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} (\mathtt{abs}\ t') \Rightarrow t \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} t'$.
$z \notin L \Rightarrow z \notin L'' \Rightarrow (\Gamma, z \mapsto \mathtt{fvar}\ x) \approx (\Gamma', z \mapsto \mathtt{fvar}\ x') \Rightarrow (\mathtt{fvar}\ x) \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} (\mathtt{fvar}\ x')$.
Therefore, for any $z \notin L$ we have $t \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} t' \wedge (\mathtt{fvar}\ x) \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} (\mathtt{fvar}\ x')$,
and by CE_OP3 (Lemma 15), $t^x \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} t'^{x'}$.
Furthermore, $z \notin L \Rightarrow z \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Gamma')$, so that $\Gamma \approx \Gamma' \overset{L32}{\Rightarrow} \Gamma \approx^{\mathtt{dom}(\Gamma) \cup \{z\}} \Gamma'$.
Finally, $\mathtt{lc}\ (\mathtt{abs}\ t) \Rightarrow \mathtt{lc}\ t^x$.
We can then conclude that $\forall z \notin L.(\Gamma, z \mapsto t^x) \approx (\Gamma', z \mapsto t'^{x'})$.
– $\overline{x} = [y' : \overline{x'}]$ with $y' \mapsto \mathtt{fvar}\ y'' \in \Gamma \wedge y' \neq y$.
Let $z \notin L$,
$z \notin L' \Rightarrow (\Gamma, z \mapsto \mathtt{abs}\ t) \ominus \overline{x} = (\Gamma \ominus y', z \mapsto \mathtt{abs}\ t[y''/y']) \ominus \overline{x'} \approx (\Gamma', z \mapsto \mathtt{abs}\ t')$;
$z \notin L'' \Rightarrow (\Gamma, z \mapsto \mathtt{fvar}\ x) \ominus \overline{x} = (\Gamma \ominus y', z \mapsto (\mathtt{fvar}\ x)[y''/y']) \ominus \overline{x'} \approx (\Gamma', z \mapsto \mathtt{fvar}\ x')$.

By induction hypothesis:
$(\Gamma \ominus y', y \mapsto (\texttt{fvar}\ x)[y''/y'], z \mapsto t[y''/y']^y) \ominus [y : \overline{x'}] \approx (\Gamma', z \mapsto t'^{x'})$
$\overset{y \neq y'}{\Rightarrow} (\Gamma, y \mapsto \texttt{fvar}\ x, z \mapsto t^y) \ominus [y' : y : \overline{x'}] \approx (\Gamma', z \mapsto t'^{x'})$
$\overset{L19}{\Rightarrow} (\Gamma, y \mapsto \texttt{fvar}\ x, z \mapsto t^y) \ominus [y : y' : \overline{x'}] \approx (\Gamma', z \mapsto t'^{x'})$
$\Rightarrow (\Gamma, y \mapsto \texttt{fvar}\ x, z \mapsto t^y) \ominus [y : \overline{x}] \approx (\Gamma', z \mapsto t'^{x'})$.

Now we check that $\texttt{ok}\ (\Gamma, y \mapsto \texttt{fvar}\ x, z \mapsto t^y)$:

$$\left.\begin{array}{l} \Gamma \gtrsim_I \Gamma'' \overset{L20}{\Rightarrow} \texttt{ok}\ \Gamma \wedge \texttt{ok}\ \Gamma' \\ y \notin \texttt{dom}(\Gamma) \\ \texttt{lc}\ (\texttt{fvar}\ x) \end{array}\right\} \Rightarrow \texttt{ok}\ (\Gamma, y \mapsto \texttt{fvar}\ x).$$

$$\left.\begin{array}{l} \texttt{ok}\ (\Gamma, y \mapsto \texttt{fvar}\ x) \\ z \notin L \supseteq \texttt{dom}(\Gamma) \cup \{y\} \\ \texttt{lc}\ (\texttt{abs}\ t) \Rightarrow \texttt{lc}\ t^y \end{array}\right\} \Rightarrow \texttt{ok}\ (\Gamma, y \mapsto \texttt{fvar}\ x, z \mapsto t^y).$$

By Proposition 4,
$\forall z \notin L. (\Gamma, y \mapsto \texttt{fvar}\ x, z \mapsto t^y) \ominus [y : \overline{x}] \approx (\Gamma', z \mapsto t'^{x'}) \Rightarrow ((\Gamma, y \mapsto \texttt{fvar}\ x) : t^y) \gtrsim_I (\Gamma' : t'^{x'})$. □

If two let expressions are indirection related, then their body terms are also related with respect to the heaps extended with the local declarations.

Lemma 40.
INTR_VARS $\quad \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma' : \texttt{let}\ \overline{t'}\ \texttt{in}\ t') \wedge$
$(\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \gtrsim_I (\Gamma' : \texttt{let}\ \overline{t'}\ \texttt{in}\ t') \Rightarrow ((\Gamma, \overline{x} \mapsto \overline{t^x}) : t^{\overline{x}}) \gtrsim_I ((\Gamma', \overline{x} \mapsto \overline{t'^x}) : t'^{\overline{x}})$.

Proof.
$(\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \gtrsim_I (\Gamma' : \texttt{let}\ \overline{t'}\ \texttt{in}\ t')$
$\Rightarrow \forall z \notin L\,.\,(\Gamma, z \mapsto \texttt{let}\ \overline{t}\ \texttt{in}\ t) \gtrsim_I (\Gamma', z \mapsto \texttt{let}\ \overline{t'}\ \texttt{in}\ t')$ for some finite $L \subseteq Id$
$\overset{C1}{\Rightarrow} (\Gamma, z \mapsto \texttt{let}\ \overline{t}\ \texttt{in}\ t) \ominus \overline{y} \approx (\Gamma', z \mapsto \texttt{let}\ \overline{t'}\ \texttt{in}\ t')$ with $\overline{y} = \texttt{dom}(\Gamma) - \texttt{dom}(\Gamma')$.
Consider any $z \notin L$ such that $\texttt{fresh}\ z\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \wedge \texttt{fresh}\ z\ \texttt{in}\ (\Gamma' : \texttt{let}\ \overline{t'}\ \texttt{in}\ t') \wedge z \notin \overline{x}$.
Now we prove by induction on the length of $\overline{y}$ that

$$(\Gamma, z \mapsto \texttt{let}\ \overline{t}\ \texttt{in}\ t) \ominus \overline{y} \approx (\Gamma', z \mapsto \texttt{let}\ \overline{t'}\ \texttt{in}\ t') \Rightarrow (\Gamma, \overline{x} \mapsto \overline{t^x}, z \mapsto t^{\overline{x}}) \ominus \overline{y} \approx (\Gamma', \overline{x} \mapsto \overline{t'^x}, z \mapsto t'^{\overline{x}})$$

– $\overline{y} = [\,]$
$\Rightarrow (\Gamma, z \mapsto \texttt{let}\ \overline{t}\ \texttt{in}\ t) \approx (\Gamma', z \mapsto \texttt{let}\ \overline{t'}\ \texttt{in}\ t')$
$\overset{L28}{\Rightarrow} \Gamma \approx^{\texttt{dom}(\Gamma) \cup \{z\}} \Gamma' \wedge \texttt{let}\ \overline{t}\ \texttt{in}\ t \approx^{\texttt{dom}(\Gamma) \cup \{z\}} \texttt{let}\ \overline{t'}\ \texttt{in}\ t' \wedge \texttt{lc}\ (\texttt{let}\ \overline{t'}\ \texttt{in}\ t') \wedge z \notin \texttt{dom}(\Gamma)$
$\texttt{let}\ \overline{t}\ \texttt{in}\ t \approx^{\texttt{dom}(\Gamma) \cup \{z\}} \texttt{let}\ \overline{t'}\ \texttt{in}\ t' \Rightarrow |\overline{t}| = |\overline{t'}| \wedge \overline{t} \approx^{\texttt{dom}(\Gamma) \cup \{z\}} \overline{t'} \wedge t \approx^{\texttt{dom}(\Gamma) \cup \{z\}} t'$
$\overset{L14}{\Rightarrow} \overline{t} \approx^{\texttt{dom}(\Gamma) \cup \overline{x} \cup \{z\}} \overline{t'} \wedge t \approx^{\texttt{dom}(\Gamma) \cup \overline{x} \cup \{z\}} t'$
$\overset{L15}{\Rightarrow} \overline{t^x} \approx^{\texttt{dom}(\Gamma) \cup \overline{x} \cup \{z\}} \overline{t'^x} \wedge t^{\overline{x}} \approx^{\texttt{dom}(\Gamma) \cup \overline{x} \cup \{z\}} t'^{\overline{x}}$.
$\texttt{lc}\ \texttt{let}\ \overline{t'}\ \texttt{in}\ t' \Rightarrow \texttt{lc}\ t^{\overline{x}} \wedge \texttt{lc}\ \overline{t}^{\overline{x}} \Rightarrow \texttt{ok}\ (\Gamma, \overline{x} \mapsto \overline{t^x}, z \mapsto t^{\overline{x}}) \wedge \texttt{ok}\ (\Gamma', \overline{x} \mapsto \overline{t'^x}, z \mapsto t'^{\overline{x}})$
$\overset{L17}{\Rightarrow} (\Gamma, \overline{x} \mapsto \overline{t^x}, z \mapsto t^{\overline{x}}) \approx (\Gamma', \overline{x} \mapsto \overline{t'^x}, z \mapsto t'^{\overline{x}})$.
– $\overline{y} = [y : \overline{z}]$ with $y \mapsto \texttt{fvar}\ y' \in \Gamma \wedge y \neq z$.
$\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \Rightarrow \texttt{fresh}\ \overline{x}\ \texttt{in}\ \Gamma \ominus y : \texttt{let}\ \overline{t}[y'/y]\ \texttt{in}\ t[y'/y]$.
$(\Gamma, z \mapsto \texttt{let}\ \overline{t}\ \texttt{in}\ t) \ominus \overline{y} = (\Gamma \ominus y, z \mapsto \texttt{let}\ \overline{t}[y'/y]\ \texttt{in}\ t[y'/y]) \ominus \overline{z} \approx (\Gamma', z \mapsto \texttt{let}\ \overline{t'}\ \texttt{in}\ t')$.
$\overset{I.H.}{\Rightarrow} (\Gamma \ominus y, \overline{x} \mapsto \overline{t}[y'/y]^{\overline{x}}, z \mapsto t[y'/y]^{\overline{x}}) \ominus \overline{z} \approx (\Gamma', \overline{x} \mapsto \overline{t'^x}, z \mapsto t'^{\overline{x}})$
$\Rightarrow (\Gamma, \overline{x} \mapsto \overline{t^x}, z \mapsto t^{\overline{x}}) \ominus \overline{y} \approx (\Gamma', \overline{x} \mapsto \overline{t'^x}, z \mapsto t'^{\overline{x}})$.

By Proposition 4 we have $(\Gamma, \overline{x} \mapsto \overline{t^x}, z \mapsto t^{\overline{x}}) \gtrsim_I (\Gamma', \overline{x} \mapsto \overline{t'^x}, z \mapsto t'^{\overline{x}})$ and the result is valid for any $z \notin L$ and sufficiently fresh, so that $((\Gamma, \overline{x} \mapsto \overline{t^x}) : t^{\overline{x}}) \gtrsim_I ((\Gamma', \overline{x} \mapsto \overline{t'^x}) : t'^{\overline{x}})$. □

We also need some auxiliary results for heap-context-equivalence.

Lemma 41.

HCE_SUBS $\quad \Gamma \approx^V \Gamma' \wedge y \notin V \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma' \Rightarrow \Gamma[y/x] \approx^{V[y/x]} \Gamma'[y/x]$

IR_SUBS $\quad \Gamma \succsim_I \Gamma' \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma' \Rightarrow \Gamma[y/x] \succsim_I \Gamma'[y/x]$

IR_HT_SUBS $\quad (\Gamma : t) \succsim_I (\Gamma' : t') \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma' : t')$
$\Rightarrow (\Gamma[y/x] : t[y/x]) \succsim_I (\Gamma'[y/x] : t'[y/x])$

Proof.

HCE_SUBS

By rule induction, where the case of the empty heap is immediate.

$$(\Theta, z \mapsto t) \approx^V (\Theta', z \mapsto t') \begin{cases} \Theta \approx^V \Theta' \overset{I.H.}{\Rightarrow} \Theta[y/x] \approx^{V[y/x]} \Theta'[y/x] \\ t \approx^V t' \overset{L15}{\Rightarrow} t[y/x] \approx^{V[y/x]} t'[y/x] \\ \texttt{lc}\ t \Rightarrow \texttt{lc}\ t[y/x] \end{cases}$$

- $x \neq z \Rightarrow z[y/x] = z \notin \texttt{dom}(\Theta) \overset{z \neq y}{\Rightarrow} z[y/x] \notin \texttt{dom}(\Theta[y/x])$.
- $x = z \Rightarrow z[y/x] = y \notin \texttt{dom}(\Theta) \overset{x \notin \texttt{dom}(\Theta)}{\Rightarrow} z[y/x] \notin \texttt{dom}(\Theta[y/x])$.

Therefore, $(\Theta[y/x], z[y/x] \mapsto t[y/x]) \approx^{V[y/x]} (\Theta'[y/x], z[y/x] \mapsto t'[y/x])$.
Thus, $\Gamma[y/x] \approx^{V[y/x]} \Gamma'[y/x]$.

IR_SUBS

By rule induction.

- $\Gamma \approx \Gamma' \Rightarrow \Gamma \approx^{\texttt{dom}(\Gamma)} \Gamma' \overset{\text{HCE\_SUBS}}{\Rightarrow} \Gamma[y/x] \approx^{\texttt{dom}(\Gamma[y/x])} \Gamma'[y/x] \Rightarrow \Gamma[y/x] \succsim_I \Gamma'[y/x]$.
- $\texttt{ok}\ \Gamma \wedge \Gamma \ominus z \succsim_I \Gamma' \wedge z \in \texttt{Ind}(\Gamma)$
  $\texttt{ok}\ \Gamma \overset{y \notin \texttt{dom}(\Gamma)}{\Rightarrow} \texttt{ok}\ \Gamma[y/x]$
  $\Gamma \ominus z \succsim_I \Gamma' \overset{I.H.}{\Rightarrow} (\Gamma \ominus z)[y/x] \succsim_I \Gamma'[y/x]$
  - $x \neq z \Rightarrow z \in \texttt{Ind}(\Gamma[y/x]) \wedge (\Gamma \ominus z)[y/x] = \Gamma[y/x] \ominus z \Rightarrow \Gamma[y/x] \succsim_I \Gamma'[y/x]$.
  - $x = z \Rightarrow y \in \texttt{Ind}(\Gamma[y/x]) \wedge (\Gamma \ominus z)[y/x] = \Gamma[y/x] \ominus y \Rightarrow \Gamma[y/x] \succsim_I \Gamma'[y/x]$.

IR_HT_SUBS

$$\begin{array}{rcl} (\Gamma : t) \succsim_I (\Gamma' : t') & \Rightarrow & \forall z \notin L\ .\ (\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t'), \text{for some finite } L \subset Id \\ & \overset{\text{IR\_SUBS}}{\Rightarrow} & \forall z \notin L \cup \{y\}\ .\ (\Gamma[y/x], \underbrace{z[y/x]}_{z'} \mapsto t[y/x]) \succsim_I (\Gamma'[y/x], \underbrace{z[y/x]}_{z'} \mapsto t'[y/x]) \\ & \Rightarrow & (\Gamma[y/x] : t[y/x]) \succsim_I (\Gamma'[y/x] : t'[y/x]) \end{array}$$

□

Proposition 7

EQ_IR_AN
$$\begin{array}{l} (\Gamma_A : t_A) \succsim_I (\Gamma_N : t_N) \wedge \\ \forall \overline{x} \notin L \subseteq Id\ .\ \Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : {w_A}^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}_A{}^{\overline{x}}) = \overline{s}_A \wedge {}^{\backslash \overline{x}}({w_A}^{\overline{x}}) = w_A \\ \Rightarrow \exists \overline{y} \notin L\ .\ \exists \overline{s}_N \subset LNExp\ .\ \exists w_N \in LNVal\ . \\ \Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : {w_N}^{\overline{z}} \wedge {}^{\backslash \overline{z}}(\overline{s}_N{}^{\overline{z}}) = \overline{s}_N \wedge {}^{\backslash \overline{z}}({w_N}^{\overline{z}}) = w_N \wedge \overline{z} \subseteq \overline{y} \\ \wedge ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : {w_A}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : {w_N}^{\overline{z}}) \end{array}$$

EQ_IR_NA
$$\begin{array}{l} (\Gamma_A : t_A) \succsim_I (\Gamma_N : t_N) \wedge \\ \forall \overline{x} \notin L \subseteq Id\ .\ \Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : {w_N}^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}_N{}^{\overline{x}}) = \overline{s}_N \wedge {}^{\backslash \overline{x}}({w_N}^{\overline{x}}) = w_N \\ \Rightarrow \exists \overline{z} \notin L\ .\ \exists \overline{s}_A \subset LNExp\ .\ \exists w_A \in LNVal\ . \\ \Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : {w_A}^{\overline{y}} \wedge {}^{\backslash \overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \wedge {}^{\backslash \overline{y}}({w_A}^{\overline{y}}) = w_A \wedge \overline{z} \subseteq \overline{y} \\ \wedge ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : {w_A}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : {w_N}^{\overline{z}}) \end{array}$$

Proof.

$$(\Gamma_A : t_A) \succsim_I (\Gamma_N : t_N) \overset{L22}{\Rightarrow} t_A \sim_S t_N.$$

EQ_IR_AN: By rule induction:

- Rule LNLam
  $t_A \equiv \texttt{abs}\ u_A \Rightarrow t_N \equiv \texttt{abs}\ u_N$, and in this case $\overline{x} = [\,]$ (so that $\overline{y} = [\,]$ too).
  It is easy to prove that for any $t \in \mathit{LNExp}$ it is $t^{[\,]} = t = {}^{\backslash [\,]}t$.
  $$\begin{array}{rcl}(\Gamma_A : \texttt{abs}\ u_A) \succsim_I (\Gamma_N : \texttt{abs}\ u_N) & \Rightarrow & \forall z \notin L' \,.\, (\Gamma_A, z \mapsto \texttt{abs}\ u_A) \succsim_I (\Gamma_N, z \mapsto \texttt{abs}\ u_N) \\ & \overset{L20}{\Rightarrow} & \texttt{ok}\ (\Gamma_N, z \mapsto \texttt{abs}\ u_N) \\ & \Rightarrow & \texttt{ok}\ \Gamma_N \wedge \texttt{lc}\ (\texttt{abs}\ u_N) \\ & \overset{\text{LNLam}}{\Rightarrow} & \Gamma_N : \texttt{abs}\ u_N \Downarrow^N \Gamma_N : \texttt{abs}\ u_N, \text{and take } \overline{z} = [\,].\end{array}$$
- Rule ALNVar.
  $t_A \equiv \texttt{fvar}\ x_A \wedge \Gamma_A = (\Gamma'_A, x_A \mapsto u_A)$.
  On the one hand, $\forall \overline{x} \notin L.(\Gamma'_A, x_A \mapsto u_A) : \texttt{fvar}\ x_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}}$
  $\Rightarrow \forall \overline{x} \notin L.(\Gamma'_A, x_A \mapsto u_A) : u_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}}$.
  On the other hand, $t_N \equiv \texttt{fvar}\ x_N$ and, by hypothesis, $((\Gamma'_A, x_A \mapsto u_A) : \texttt{fvar}\ x_A) \succsim_I (\Gamma_N : \texttt{fvar}\ x_N)$
  $\overset{L34}{\Rightarrow} \Gamma_N = (\Gamma'_N, x_N \mapsto u_N)$
  $\overset{L36}{\Rightarrow} ((\Gamma'_A, x_A \mapsto u_A) : u_A) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : u_N) \vee$
  $((\Gamma'_A, x_A \mapsto u_A) : u_A) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : \texttt{fvar}\ x_N)$.
  In the first case, by induction hypothesis we have $(\Gamma'_N, x_N \mapsto u_N) : u_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}}$
  and $((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$, for some $\overline{z} \subseteq \overline{y} \notin L$, $\overline{s}_N$ and w_N.
  By applying rule ALNVar to this result we obtain $\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}}$.
  In the second case, by induction hypothesis we obtain directly
  $(\Gamma'_N, x_N \mapsto u_N) : \texttt{fvar}\ x_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}}$,
  with $((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$ for some $\overline{z} \subseteq \overline{y} \notin L$, $\overline{s}_N$ and w_N.
- Rule ALNApp.
  $t_A \equiv \texttt{app}\ t'_A\ (\texttt{fvar}\ x_A) \Rightarrow t_N \equiv \texttt{app}\ t'_N\ (\texttt{fvar}\ x_N)$.
  By hypothesis, $\forall \overline{x} \notin L \,.\, \Gamma_A : \texttt{app}\ t'_A\ (\texttt{fvar}\ x_A) \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}}$.
  Let us choose a particular $\overline{x} \notin L$ such that $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma_A : t_A)$;
  $\overline{x}$ can be decomposed as $\overline{x} = [z : \overline{x}_1 +\!\!+ \overline{x}_2]$ with

  $$\Gamma_A : t'_A \Downarrow^A (\Gamma_A, \overline{x}_1 \mapsto \overline{s}_{1A}{}^{\overline{x}_1}) : \texttt{abs}\ u'_A{}^{\overline{x}_1} \tag{3}$$

  and

  $$(\Gamma_A, \overline{x}_1 \mapsto \overline{s}_{1A}{}^{\overline{x}_1}, z \mapsto \texttt{fvar}\ x_A) : (u'_A{}^{\overline{x}_1})^z \Downarrow^A (\Gamma_A, \overline{x}_1 \mapsto \overline{s}_{1A}{}^{\overline{x}_1}, z \mapsto \texttt{fvar}\ x_A, \overline{x}_2 \mapsto \overline{s}_{2A}{}^{\overline{x}_2}) : w'_A{}^z \tag{4}$$

  where $w'_A{}^z = w_A{}^{\overline{x}}$.
  Let $L' = L \cup \texttt{names}(\Gamma_A : t_A) \cup \texttt{fv}(\overline{s}_{1A}) \cup \texttt{fv}(u'_A) \cup \{x_N\}$; by applying RENAMING2 (Lemma 12) to (3) we obtain
  $$\forall \overline{x}_1 \notin L' \,.\, \Gamma_A : t'_A \Downarrow^A (\Gamma_A, \overline{x}_1 \mapsto \overline{s}_{1A}{}^{\overline{x}_1}) : \texttt{abs}\ u'_A{}^{\overline{x}_1}.$$
  By hypothesis, $(\Gamma_A : \texttt{app}\ t'_A\ (\texttt{fvar}\ x_A)) \succsim_I (\Gamma_N : \texttt{app}\ t'_N\ (\texttt{fvar}\ x_N)) \overset{L37}{\Rightarrow} (\Gamma_A : t'_A) \succsim_I (\Gamma_N : t'_N)$, so that by induction hypothesis

  $$\Gamma_N : t'_N \Downarrow^N (\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : w'_N{}^{\overline{z}_1} \ \wedge\ ((\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}) : \texttt{abs}\ u'_A{}^{\overline{y}_1}) \succsim_I ((\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : w'_N{}^{\overline{z}_1})$$

  for some $\overline{y}_1 \notin L'$, and with $\overline{z}_1 \subseteq \overline{y}_1$. It must be that $w'_N{}^{\overline{z}_1} \equiv \texttt{abs}\ u_N$ for some term u_N.
  Now we apply RENAMING1 (Lemma 12) to (4) to obtain

  $$(\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, z \mapsto \texttt{fvar}\ x_A) : (u'_A{}^{\overline{y}_1})^z \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, z \mapsto \texttt{fvar}\ x_A, \overline{x}_2 \mapsto \overline{s}_{2A}{}^{\overline{x}_2}) : w''_A{}^z$$

where ${w''_A}^z = {w_A}^{[z:\overline{y}_1 +\!\!+ \overline{x}_2]}$.
Let $L'' = L' \cup \mathtt{fv}(\overline{s}_{2A}) \cup \mathtt{fv}(w''_A) \cup \overline{y}_1 \cup \{z\}$; by RENAMING2 (Lemma 12)

$$\forall \overline{x}_2 \notin L'' \,.\, (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, z \mapsto \mathtt{fvar}\ x_A) : ({u'_A}^{\overline{y}_1})^z \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, z \mapsto \mathtt{fvar}\ x_A, \overline{x}_2 \mapsto \overline{s}_{2A}{}^{\overline{x}_2}) : {w''_A}^z$$

By Lemma 37 we also have $(\Gamma_A : \mathtt{fvar}\ x_A) \gtrsim_I (\Gamma_N : \mathtt{fvar}\ x_N)$.
If $\Delta_A = (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1})$ and $\Delta_N = (\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1})$ then $\Gamma_A \subseteq \Delta_A \wedge \Gamma_N \subseteq \Delta_N \wedge \Delta_A \gtrsim_I \Delta_N$, by Lemmas 9 and 22.
Moreover, since $x_A \notin \overline{y}_1$ it is verified that $(x_A \notin \mathtt{dom}(\Gamma_A) \Rightarrow x_A \notin \mathtt{dom}(\Delta_A))$;
likewise $x_N \notin \overline{y}_1 \Rightarrow x_N \notin \overline{z}_1$ so that it is verified also that $(x_N \notin \mathtt{dom}(\Gamma_N) \Rightarrow x_N \notin \mathtt{dom}(\Delta_N))$.
Furthermore, $y \in \mathtt{dom}(\Gamma_A) \Rightarrow y \notin \overline{y}_1 \Rightarrow y \notin \overline{z}_1$, thus, $(y \in \mathtt{dom}(\Gamma_A) \wedge y \notin \mathtt{dom}(\Gamma_N) \Rightarrow y \notin \mathtt{dom}(\Delta_N))$.
Therefore, all the conditions of Lemma 38 are satisfied and we infer that
$(\Delta_A : \mathtt{fvar}\ x_A) \gtrsim_I (\Delta_N : \mathtt{fvar}\ x_N) \overset{L39}{\Rightarrow} ((\Delta_A, z \mapsto \mathtt{fvar}\ x_A) : ({u'_A}^{\overline{y}_1})^z) \gtrsim_I (\Delta_N : {u_N}^{x_N})$.
By induction hypothesis

$$(\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : {u_N}^{x_N} \Downarrow^N (\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}, \overline{z}_2 \mapsto \overline{s}_{2N}{}^{\overline{z}_2}) : {w''_N}^{\overline{z}_2} \tag{5}$$

and

$$((\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, z \mapsto \mathtt{fvar}\ x_A, \overline{y}_2 \mapsto \overline{s}_{2A}{}^{\overline{y}_2}) : {w_A}^{[z:\overline{y}_1 +\!\!+ \overline{y}_2]}) \gtrsim_I ((\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}, \overline{z}_2 \mapsto \overline{s}_{2N}{}^{\overline{z}_2}) : {w''_N}^{\overline{z}_2})$$

for some $\overline{y}_2 \notin L''$, and with $\overline{z}_2 \subseteq \overline{y}_2$.
Let $\overline{y} = [z : \overline{y}_1 +\!\!+ \overline{y}_2]$ and $\overline{z} = \overline{z}_1 +\!\!+ \overline{z}_2$ (notice that $\overline{z} \subseteq \overline{y}$), then we can rewrite (5) as

$$(\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : {u_N}^{x_N} \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : {w_N}^{\overline{z}}$$

with $((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : {w_A}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : {w_N}^{\overline{z}})$.
Finally, by the rule LNAPP the required result is obtained.

– Rule LNLET.
$t_A \equiv \mathtt{let}\ \overline{t}_A\ \mathtt{in}\ t'_A \Rightarrow t_N \equiv \mathtt{let}\ \overline{t}_N\ \mathtt{in}\ t'_N \wedge |\overline{t}_A| = |\overline{t}_N|$.
By hypothesis, $\forall \overline{x} \notin L \,.\, \Gamma_A : \mathtt{let}\ \overline{t}_A\ \mathtt{in}\ t'_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : {w_A}^{\overline{x}}$.
Let us choose a particular $\overline{x} \notin L$ such that $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ (\Gamma_A : \mathtt{let}\ \overline{t}_A\ \mathtt{in}\ t'_A)$, and we decompose the set of names as $\overline{x} = \overline{x}_1 +\!\!+ \overline{x}_2$, with $|\overline{x}_1| = |\overline{t}_A|$.
Then, by the LNLET rule we have that

$$\forall \overline{y_1}^{|\overline{t}_A|} \notin L'.(\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}) : {t'_A}^{\overline{y}_1} \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}, \overline{x}_2 \mapsto \overline{s'}_A{}^{\overline{y}_1}) : {w'_A}^{\overline{y}_1},$$

for some finite $L' \subseteq Id$, with $\overline{x}_1 \notin L'$ and $\overline{s}_A{}^{\overline{x}} = \overline{t}_A{}^{\overline{y}_1} +\!\!+ \overline{s'}_A{}^{\overline{y}_1} \wedge {w_A}^{\overline{x}} = {w'_A}^{\overline{y}_1}$.
Let $L'' = L \cup L' \cup \mathtt{names}(\Gamma_A : t_A) \cup \mathtt{names}(\Gamma_N : t_N)$, and fix a particular $\overline{y}_1 \notin L''$; thanks to Lemma 6, $\overline{s'}_A{}^{\overline{y}_1}$ can be expressed as $\overline{s''}_A{}^{\overline{x}_2}$ and ${w'_A}^{\overline{y}_1}$ as ${w''_A}^{\overline{x}_2}$, with $^{\backslash \overline{x}_2}(\overline{s''}_A{}^{\overline{x}_2}) = \overline{s''}_A \wedge {}^{\backslash \overline{x}_2}({w''_A}^{\overline{x}_2}) = w''_A$, so that

$$(\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}) : {t'_A}^{\overline{y}_1} \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}, \overline{x}_2 \mapsto \overline{s''}_A{}^{\overline{x}_2}) : {w''_A}^{\overline{x}_2}.$$

By RENAMING2 (Lemma 12) we can obtain

$$\forall \overline{y}_2 \notin L'' \cup \overline{y}_1.(\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}) : {t'_A}^{\overline{y}_1} \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}, \overline{y}_2 \mapsto \overline{s''}_A{}^{\overline{y}_2}) : {w''_A}^{\overline{y}_2}.$$

Moreover, by Lemma 40, $((\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}) : {t'_A}^{\overline{y}_1}) \gtrsim_I ((\Gamma_N, \overline{y}_1 \mapsto \overline{t}_N{}^{\overline{y}_1}) : {t'_N}^{\overline{y}_1})$.
Hence, by induction hypothesis there exists $\overline{y}_2 \notin L'' \cup \overline{y}_1$, and $\overline{s''}_N$ and w''_N such that

$$(\Gamma_N, \overline{y}_1 \mapsto \overline{t}_N{}^{\overline{y}_1}) : {t'_N}^{\overline{y}_1} \Downarrow^N (\Gamma_N, \overline{y}_1 \mapsto \overline{t}_N{}^{\overline{y}_1}, \overline{z}_2 \mapsto \overline{s''}_N{}^{\overline{z}_2}) : {w''_N}^{\overline{z}_2}$$

with ${}^{\backslash\overline{z}_2}(\overline{s''}_N{}^{\overline{z}_2}) = \overline{s''}_N \wedge {}^{\backslash\overline{z}_2}(w''_N{}^{\overline{z}_2}) = w''_N \wedge \overline{z}_2 \subseteq \overline{y}_2$,
and $(\Gamma_A, \overline{y}_1 \mapsto \overline{t}_A{}^{\overline{y}_1}, \overline{y}_2 \mapsto \overline{s''}_A{}^{\overline{y}_2}) : w''_A{}^{\overline{y}_2}) \succsim_I ((\Gamma_N, \overline{y}_1 \mapsto \overline{t}_N{}^{\overline{y}_1}, \overline{z}_2 \mapsto \overline{s''}_N{}^{\overline{z}_2}) : w''_N{}^{\overline{z}_2})$.
Notice that $\overline{y}_1 +\!\!+ \overline{y}_2 \notin L$.
By Lemma 6, $\overline{t}_N{}^{\overline{y}_1} +\!\!+ \overline{s''}_N{}^{\overline{z}_2}$ can be expressed as $\overline{s'}_N{}^{\overline{y}_1}$ and $w''_N{}^{\overline{z}_2}$ as $w'_N{}^{\overline{y}_1}$ with ${}^{\backslash\overline{y}_1}(\overline{s'}_N{}^{\overline{y}_1}) = \overline{s'}_N \wedge {}^{\backslash\overline{y}_1}(w'_N{}^{\overline{y}_1}) = w'_N$.
Let $L''' = L'' \cup \mathtt{fv}(\overline{s}_A) \cup \mathtt{fv}(w_A) \cup \mathtt{fv}(\overline{s'}_N) \cup \mathtt{fv}(w'_N) \cup \overline{y}_1 +\!\!+ \overline{y}_2 \cup \overline{z}_2$;
by RENAMING1 (Lemma 12) we obtain

$$\forall \overline{z}_1 \notin L'''.(\Gamma_N, \overline{z}_1 \mapsto \overline{t}_N{}^{\overline{z}_1}) : t'_N{}^{\overline{z}_1} \Downarrow^N (\Gamma_N, \overline{z}_1 +\!\!+ \overline{z}_2 \mapsto \overline{s'}_N{}^{\overline{z}_1}) : w'_N{}^{\overline{z}_1}.$$

Then, by the LNLET rule we infer that

$$\Gamma_N : \mathtt{let}\ \overline{t}_N\ \mathtt{in}\ t'_N \Downarrow^N (\Gamma_N, \overline{z}_1 +\!\!+ \overline{z}_2 \mapsto \overline{s'}_N{}^{\overline{z}_1}) : w'_N{}^{\overline{z}_1}, \text{for some } \overline{z}_1 \notin L'''.$$

Once again, by Lemma 6, we rewrite $\overline{s'}_N{}^{\overline{z}_1} = \overline{s}_N{}^{\overline{z}_1 +\!\!+ \overline{z}_2}$ and $w'_N{}^{\overline{z}_1} = w_N{}^{\overline{z}_1 +\!\!+ \overline{z}_2}$.
Besides, notice that $\overline{t}_A{}^{\overline{y}_1} +\!\!+ \overline{s''}_A{}^{\overline{y}_2} = \overline{s}_A{}^{\overline{z}_1 +\!\!+ \overline{y}_2}$ and $w''_A{}^{\overline{y}_2} = w_A{}^{\overline{y}_1 +\!\!+ \overline{y}_2}$; hence
$((\Gamma_A, \overline{y}_1 +\!\!+ \overline{y}_2 \mapsto \overline{s}_A{}^{\overline{y}_1 +\!\!+ \overline{y}_2}) : w_A{}^{\overline{y}_1 +\!\!+ \overline{y}_2}) \succsim_I ((\Gamma_N, \overline{y}_1 +\!\!+ \overline{z}_2 \mapsto \overline{s}_N{}^{\overline{y}_1 +\!\!+ \overline{z}_2}) : w_N{}^{\overline{y}_1 +\!\!+ \overline{z}_2})$
$\overset{L41}{\Rightarrow} (\Gamma_A, \overline{z}_1 +\!\!+ \overline{y}_2 \mapsto \overline{s}_A{}^{\overline{z}_1 +\!\!+ \overline{y}_2}) : w_A{}^{\overline{z}_1 +\!\!+ \overline{y}_2}) \succsim_I ((\Gamma_N, \overline{z}_1 +\!\!+ \overline{z}_2 \mapsto \overline{s}_N{}^{\overline{z}_1 +\!\!+ \overline{z}_2}) : w_N{}^{\overline{z}_1 +\!\!+ \overline{z}_2})$,
and notice that $\overline{z}_1 +\!\!+ \overline{z}_2 \subseteq \overline{z}_1 +\!\!+ \overline{y}_2 \notin L$.

EQ_IR_NA: By rule induction.

For rules LNLAM and LNLET we follow similar reasonings as in EQ_IR_AN. Therefore, we detail only the cases of rules ALNVAR and LNAPP:

– Rule ALNVAR.
$t_N \equiv \mathtt{fvar}\ x_N \wedge \Gamma_N = (\Gamma'_N, x_N \mapsto u_N)$.
On the one hand, $\forall \overline{x} \notin L.(\Gamma'_N, x_N \mapsto u_N) : \mathtt{fvar}\ x_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}}$
$\Rightarrow \forall \overline{x} \notin L.(\Gamma'_N, x_N \mapsto u_N) : u_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}}$.
On the other hand, $t_A \equiv \mathtt{fvar}\ x_A$ and, by hypothesis, $(\Gamma_A : \mathtt{fvar}\ x_A) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : \mathtt{fvar}\ x_N)$
$\overset{L34}{\Rightarrow} \Gamma_A = (\Gamma'_A, x_A \mapsto u_A) \wedge$
$\exists x_0, \ldots, x_n \in Id\,.\, x_0 = x_A \wedge x_n = x_N \wedge \forall i : 0 \leq i < n\,.\, x_i \mapsto \mathtt{fvar}\ x_{i+1} \in (\Gamma'_A, x_A \mapsto u_A)$.
Now we proceed by induction on n to prove that $(\Gamma'_A, x_A \mapsto u_A) : u_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$
and $((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$ for some $\overline{z} \subseteq \overline{y} \notin L$.
By applying rule ALNVAR to this result we obtain $\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$.
- $n = 0 \Rightarrow x_A = x_N$
$\Rightarrow ((\Gamma'_A, x_A \mapsto u_A) : \mathtt{fvar}\ x_A) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : \mathtt{fvar}\ x_A)$
$\overset{L22}{\Rightarrow} (\Gamma'_A, x_A \mapsto u_A) \succsim_I (\Gamma'_N, x_N \mapsto u_N)$
$\overset{L35}{\Rightarrow} ((\Gamma'_A, x_A \mapsto u_A) : u_A) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : u_N)$.
By (rule) induction hypothesis we are done.
- $n > 0 \Rightarrow ((\Gamma'_A, x_A \mapsto \mathtt{fvar}\ x_1) : \mathtt{fvar}\ x_A) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : \mathtt{fvar}\ x_N)$
$\overset{L36}{\Rightarrow} ((\Gamma'_A, x_A \mapsto \mathtt{fvar}\ x_1) : \mathtt{fvar}\ x_1) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : u_N) \vee$
$((\Gamma'_A, x_A \mapsto \mathtt{fvar}\ x_1) : \mathtt{fvar}\ x_1) \succsim_I ((\Gamma'_N, x_N \mapsto u_N) : \mathtt{fvar}\ x_N)$.
In the first case, by (rule) induction hypothesis we are done.
In the second case, by Lemma 34, we have $x_1 \in \mathtt{dom}(\Gamma_A)$, so that $\Gamma_A = (\Gamma''_A, x_1 \mapsto u_1)$.
Since the path from x_1 to x_N is of length $n-1$, the induction hypothesis indicates that there exists $\overline{z} \subseteq \overline{y} \notin L$, such that
$(\Gamma''_A, x_1 \mapsto u_1) : u_1 \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$ and $((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \succsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$.
By applying rule ALNVAR, we obtain $(\Gamma''_A, x_1 \mapsto u_1) : \mathtt{fvar}\ x_1 \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$, i.e.,
$(\Gamma'_A, x_A \mapsto \mathtt{fvar}\ x_1) : \mathtt{fvar}\ x_1 \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$.

– Rule LNApp.
$t_N \equiv \texttt{app}\ t'_N\ (\texttt{fvar}\ x_N) \Rightarrow t_A \equiv \texttt{app}\ t'_A\ (\texttt{fvar}\ x_A)$.
By hypothesis, $\forall \overline{x} \notin L\,.\,\Gamma_N : \texttt{app}\ t'_N\ (\texttt{fvar}\ x_N) \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}}$.
Let us choose a particular $\overline{x} \notin L$ such that $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma_N : t_N)$;
$\overline{x}$ can be decomposed as $\overline{x} = \overline{x}_1 +\!\!+ \overline{x}_2$ with

$$\Gamma_N : t'_N \Downarrow^N (\Gamma_N, \overline{x}_1 \mapsto \overline{s}_{1N}{}^{\overline{x}_1}) : \texttt{abs}\ u'_N{}^{\overline{x}_1} \tag{6}$$

and

$$(\Gamma_N, \overline{x}_1 \mapsto \overline{s}_{1N}{}^{\overline{x}_1}) : (u'_N{}^{\overline{x}_1})^{x_N} \Downarrow^N (\Gamma_N, \overline{x}_1 \mapsto \overline{s}_{1N}{}^{\overline{x}_1}, \overline{x}_2 \mapsto \overline{s}_{2N}{}^{\overline{x}_2}) : w'_N{}^{\overline{x}_2} \tag{7}$$

where $w'_N{}^{\overline{x}_2} = w_N{}^{\overline{x}}$.
Let $L' = L \cup \texttt{names}(\Gamma_N : t_N) \cup \texttt{fv}(\overline{s}_{1N}) \cup \texttt{fv}(u'_N)$; by applying RENAMING2 (Lemma 12) to (6) we obtain

$$\forall \overline{x}_1 \notin L'\,.\,\Gamma_N : t'_N \Downarrow^N (\Gamma_N, \overline{x}_1 \mapsto \overline{s}_{1N}{}^{\overline{x}_1}) : \texttt{abs}\ u'_N{}^{\overline{x}_1}.$$

By hypothesis, $(\Gamma_A : \texttt{app}\ t'_A\ (\texttt{fvar}\ x_A)) \precsim_I (\Gamma_N : \texttt{app}\ t'_N\ (\texttt{fvar}\ x_N)) \overset{L37}{\Rightarrow} (\Gamma_A : t'_A) \precsim_I (\Gamma_N : t'_N)$, so that by induction hypothesis

$$\Gamma_A : t'_A \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}) : w'_A{}^{\overline{y}_1} \wedge ((\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}) : w'_A{}^{\overline{y}_1}) \precsim_I ((\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : \texttt{abs}\ u'_N{}^{\overline{z}_1})$$

for some $\overline{z}_1 \notin L'$, and with $\overline{z}_1 \subseteq \overline{y}_1$. It must be that $w'_A{}^{\overline{y}_1} \equiv \texttt{abs}\ u_A$ for some term u_A.
Now we apply RENAMING1 (Lemma 12) to (7) to obtain

$$(\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : (u'_N{}^{\overline{z}_1})^{x_N} \Downarrow^N (\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}, \overline{x}_2 \mapsto \overline{s}_{2N}{}^{\overline{x}_2}) : w''_N{}^{\overline{x}_2}$$

where $w''_N{}^{\overline{x}_2} = w_N{}^{\overline{z}_1 +\!\!+ \overline{x}_2}$.
Let $L'' = L' \cup \texttt{fv}(\overline{s}_{2N}) \cup \texttt{fv}(w''_N) \cup \overline{z}_1$; by RENAMING2 (Lemma 12)

$$\forall \overline{x}_2 \notin L''\,.\,(\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}) : (u'_N{}^{\overline{z}_1})^{x_N} \Downarrow^N (\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}, \overline{x}_2 \mapsto \overline{s}_{2N}{}^{\overline{x}_2}) : w''_N{}^{\overline{x}_2}$$

By Lemma 37 we also have $(\Gamma_A : \texttt{fvar}\ x_A) \precsim_I (\Gamma_N : \texttt{fvar}\ x_N)$.
If $\Delta_A = (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1})$ and $\Delta_N = (\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1})$ then $\Gamma_A \subseteq \Delta_A \wedge \Gamma_N \subseteq \Delta_N \wedge \Delta_A \precsim_I \Delta_N$, by Lemmas 9 and 22.
On the one hand, $x_A \notin \texttt{dom}(\Gamma_A) \overset{L11}{\Rightarrow} x_A \notin \texttt{dom}(\Delta_A)$; on the other hand, since $x_N \notin \overline{z}_1$ it is verified that $(x_N \notin \texttt{dom}(\Gamma_N) \Rightarrow x_N \notin \texttt{dom}(\Delta_N))$.
Furthermore, $y \in \texttt{dom}(\Gamma_A) \Rightarrow y \notin \overline{y}_1 \Rightarrow y \notin \overline{z}_1$, thus, $(y \in \texttt{dom}(\Gamma_A) \wedge y \notin \texttt{dom}(\Gamma_N) \Rightarrow y \notin \texttt{dom}(\Delta_N))$.
Therefore, all the conditions of Lemma 38 are satisfied and we infer that
$(\Delta_A : \texttt{fvar}\ x_A) \precsim_I (\Delta_N : \texttt{fvar}\ x_N)$
$\overset{L39}{\Rightarrow} ((\Delta_A, y \mapsto \texttt{fvar}\ x_A) : u_A{}^y) \precsim_I (\Delta_N : (u'_N{}^{\overline{z}_1})^{x_N})$,
for any $y \in Id$ such that $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma_A : u_A) \wedge y \neq x_A$.
By induction hypothesis

$$(\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, y \mapsto \texttt{fvar}\ x_A) : u_A{}^y \Downarrow^A (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, y \mapsto \texttt{fvar}\ x_A, \overline{y}_2 \mapsto \overline{s}_{2A}{}^{\overline{y}_2}) : w''_A{}^{\overline{y}_2} \tag{8}$$

and

$$((\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, z \mapsto \texttt{fvar}\ x_A, \overline{y}_2 \mapsto \overline{s}_{2A}{}^{\overline{y}_2}) : w''_A{}^{\overline{y}_2}) \precsim_I ((\Gamma_N, \overline{z}_1 \mapsto \overline{s}_{1N}{}^{\overline{z}_1}, \overline{z}_2 \mapsto \overline{s}_{2N}{}^{\overline{z}_2}) : w''_N{}^{\overline{z}_2})$$

for some $\overline{z}_2 \notin L''$, and with $\overline{z}_2 \subseteq \overline{y}_2$.
Let $\overline{y} = [y : \overline{y}_1 +\!\!+ \overline{y}_2]$ and $\overline{z} = \overline{z}_1 +\!\!+ \overline{z}_2$ (notice that $\overline{z} \subseteq \overline{y}$), then we can rewrite (8) as

$$(\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, y \mapsto \texttt{fvar}\ x_A) : u_A{}^y \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$$

with $((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \precsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$.
Finally, let $L''' = \mathtt{names}(\Gamma_A) \cup \mathtt{fv}(\overline{s}_A) \cup \mathtt{fv}(u_A) \cup \mathtt{fv}(w_A) \cup \{x_A\} \cup \overline{y}$, we apply RENAMING1 (Lemma 12) to (8) to obtain

$$\forall y \notin L''' \,.\, (\Gamma_A, \overline{y}_1 \mapsto \overline{s}_{1A}{}^{\overline{y}_1}, y \mapsto \mathtt{fvar}\ x_A) : u_A{}^y \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}$$

so that the required result is obtained by the rule ALNAPP.

□

Lemma 42.

IRHT_REF $\quad \mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \Rightarrow (\Gamma : t) \precsim_I (\Gamma : t)$

Proof.

$\mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \Rightarrow \forall z \notin \mathtt{dom}(\Gamma) \,.\, \mathtt{ok}\ (\Gamma, z \mapsto t) \overset{P5}{\Rightarrow} \forall z \notin \mathtt{dom}(\Gamma) \,.\, (\Gamma, z \mapsto t) \precsim_I (\Gamma, z \mapsto t) \Rightarrow (\Gamma : t) \precsim_I (\Gamma : t)$.

□

Theorem 1 (Equivalence)

EQ_AN $\quad \Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in \mathit{LNHeap} \,.\, \exists w_N \in \mathit{LNVal} \,.\, \Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \precsim_I (\Delta_N : w_N)$

EQ_NA $\quad \Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in \mathit{LNHeap} \,.\, \exists w_A \in \mathit{LNVal} \,.\, \exists \overline{x} \subseteq \mathtt{dom}(\Delta_N) - \mathtt{dom}(\Gamma) \,.\, \exists \overline{y} \subseteq \mathit{Id} \,.\, |\overline{x}| = |\overline{y}| \wedge$
$\Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \precsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$

Proof.

EQ_AN

Assume $\Gamma : t \Downarrow^A \Delta_A : w_A$, then by Lemmas 9 and 6 the final heap and value can be written as $\Delta_A = (\Gamma, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}})$ and $w_A = w'_A{}^{\overline{x}}$ with $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ \overline{s}_A$ and $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ w'_A$.
Let $L = \mathtt{names}(\Gamma : t) \cup \mathtt{names}(\Delta_A : w_A) = \mathtt{names}(\Gamma : t) \cup \overline{x} \cup \mathtt{fv}(\overline{s}_A) \cup \mathtt{fv}(w'_A)$,
then by RENAMING2 (Lemma 12):

$$\forall \overline{y} \notin L . \Gamma : t \Downarrow^A \Delta_A[\overline{y}/\overline{x}] : w_A[\overline{y}/\overline{x}]$$

that is

$$\forall \overline{y} \notin L . \Gamma : t \Downarrow^A (\Gamma, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}})[\overline{y}/\overline{x}] : w'_A{}^{\overline{x}}[\overline{y}/\overline{x}].$$

But $\overline{x} = \mathtt{dom}(\Delta_A) - \mathtt{dom}(\Gamma) \overset{L11}{\Rightarrow} \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ (\Gamma : t)$, so that $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ \Gamma, \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ \overline{s}_A$ and $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ w'_A$.
Therefore,

$$\forall \overline{y} \notin L . \Gamma : t \Downarrow^A (\Gamma, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w'_A{}^{\overline{y}},$$

with $\backslash^{\overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \wedge \backslash^{\overline{y}}(w'_A{}^{\overline{y}}) = w'_A$ (by Lemma 3).
Moreover, by REGULARITY (Lemma 7) we have $\mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \overset{L42}{\Rightarrow} (\Gamma : t) \precsim_I (\Gamma : t)$.
By Proposition 7,

$$\exists \Delta_N . \exists w_N . \Gamma : t \Downarrow^N \Delta_N : w_N \wedge ((\Gamma, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w'_A{}^{\overline{y}}) \precsim_I (\Delta_N : w_N), \text{for some } \overline{y} \notin L,$$

with $(\Delta_N : w_N) = ((\Delta, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w'_N{}^{\overline{z}})$, where $\mathtt{fresh}\ \overline{z}\ \mathtt{in}\ \overline{s}_N, \mathtt{fresh}\ \overline{z}\ \mathtt{in}\ w'_N$ and $\overline{z} \subseteq \overline{y}$.
From $\overline{y} \notin L$ and $\overline{x} \subseteq L$ we infer that $\overline{y}$ and $\overline{x}$ are disjoint. $\overline{z}$ and $\overline{x}$ are disjoint too.
Hence, $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ ((\Gamma, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w'_A{}^{\overline{y}}) \overset{L20}{\Rightarrow} \overline{x} \notin \mathtt{dom}(\Delta_N)$.
The later result, together with $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ (\Gamma : t) \overset{L10}{\Rightarrow} \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ (\Delta_N : w_N)$.
Now by RENAMING3 (Corollary 2): $\Gamma : t \Downarrow^N \Delta_N[\overline{x}/\overline{y}] : w_N[\overline{x}/\overline{y}]$.
Besides, by Lemma 41,

$(\Delta_A : w_A) = ((\Gamma, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w'_A{}^{\overline{x}}) = ((\Gamma, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}})[\overline{x}/\overline{y}] : w'_A{}^{\overline{y}}[\overline{x}/\overline{y}]) \succsim_I (\Delta_N[\overline{x}/\overline{y}] : w_N[\overline{x}/\overline{y}])$.
Therefore, for $(\Delta'_N : w'_N) = (\Delta_N[\overline{x}/\overline{y}] : w_N[\overline{x}/\overline{y}])$ we have:

$$\Gamma : t \Downarrow^N \Delta'_N : w'_N \wedge (\Delta_A : w_A) \succsim_I (\Delta'_N : w'_N).$$

EQ_NA
Assume $\Gamma : t \Downarrow^N \Delta_N : w_N$, then by Lemmas 9 and 6 the final heap and value can be written as $\Delta_N = (\Gamma, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}})$ and $w_N = w'_N{}^{\overline{x}}$ with $\texttt{fresh}\ \overline{x}\ \texttt{in}\ \overline{s}_N$ and $\texttt{fresh}\ \overline{x}\ \texttt{in}\ w'_N$.
Let $L = \texttt{names}(\Gamma : t) \cup \texttt{names}(\Delta_N : w_N) = \texttt{names}(\Gamma : t) \cup \overline{x} \cup \texttt{fv}(\overline{s}_N) \cup \texttt{fv}(w'_N)$,
then by RENAMING2 (Lemma 12):

$$\forall \overline{y} \notin L . \Gamma : t \Downarrow^N \Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}].$$

But $\overline{x} = \texttt{dom}(\Delta_N) - \texttt{dom}(\Gamma) \overset{L11}{\Rightarrow} \texttt{fresh}\ \overline{x}\ \texttt{in}\ \Gamma$, so that $\texttt{fresh}\ \overline{x}\ \texttt{in}\ \Gamma, \texttt{fresh}\ \overline{x}\ \texttt{in}\ \overline{s}_N$ and $\texttt{fresh}\ \overline{x}\ \texttt{in}\ w'_N$.
Therefore,

$$\forall \overline{y} \notin L . \Gamma : t \Downarrow^N (\Gamma, \overline{y} \mapsto \overline{s}_N{}^{\overline{y}}) : w'_N{}^{\overline{y}},$$

with $\backslash^{\overline{y}}(\overline{s}_N{}^{\overline{y}}) = \overline{s}_N \wedge \backslash^{\overline{y}}(w'_N{}^{\overline{y}}) = w'_N$ (by Lemma 3).
Moreover, by REGULARITY (Lemma 7) we have $\texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \overset{L42}{\Rightarrow} (\Gamma : t) \succsim_I (\Gamma : t)$.
By Proposition 7,

$$\exists \Delta_A . \exists w_A . \Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \succsim_I ((\Gamma, \overline{y} \mapsto \overline{s}_N{}^{\overline{y}}) : w'_N{}^{\overline{y}}), \text{for some } \overline{y} \notin L.$$

But $((\Gamma, \overline{y} \mapsto \overline{s}_N{}^{\overline{y}}) : w'_N{}^{\overline{y}}) = ((\Gamma, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}})[\overline{y}/\overline{x}] : w'_N{}^{\overline{x}}[\overline{y}/\overline{x}]) = (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$.

□

7.6 List of Theorems, Propositions, Lemmas and Corollaries

Theorem 1.

EQ_AN $\Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in \mathit{LNHeap} . \exists w_N \in \mathit{LNVal} . \Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \succsim_I (\Delta_N : w_N)$

EQ_NA $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in \mathit{LNHeap} . \exists w_A \in \mathit{LNVal} . \exists \overline{x} \subseteq \texttt{dom}(\Delta_N) - \texttt{dom}(\Gamma) . \exists \overline{y} \subseteq \mathit{Id} . |\overline{x}| = |\overline{y}| \wedge$
$\Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \succsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$

Proposition 1.

SS_REF $t \sim_S t$
SS_SIM $t \sim_S t' \Rightarrow t' \sim_S t$
SS_TRANS $t \sim_S t' \wedge t \sim_S t'' \Rightarrow t \sim_S t''$

Proposition 2.

CE_REF $t \approx^V t$
CE_SYM $t \approx^V t' \Rightarrow t' \approx^V t$
CE_TRANS $t \approx^V t' \wedge t' \approx^V t'' \Rightarrow t \approx^V t''$

Proposition 3.

HCE_REF $\texttt{ok}\ \Gamma \Rightarrow \Gamma \approx^V \Gamma$
HCE_SYM $\Gamma \approx^V \Gamma' \Rightarrow \Gamma' \approx^V \Gamma$
HCE_TRANS $\Gamma \approx^V \Gamma' \wedge \Gamma' \approx^V \Gamma'' \Rightarrow \Gamma \approx^V \Gamma''$

Proposition 4.

IR_ALT $\Gamma \succsim_I \Gamma' \Leftrightarrow \texttt{ok}\ \Gamma \wedge \exists\ \overline{x} \subseteq \texttt{Ind}(\Gamma)\ .\ \Gamma \ominus \overline{x} \approx \Gamma'$

Proposition 5.

IR_REF $\quad \mathtt{ok}\ \Gamma \Rightarrow \Gamma \gtrsim_I \Gamma$

IR_TRANS $\quad \Gamma \gtrsim_I \Gamma' \wedge \Gamma' \gtrsim_I \Gamma'' \Rightarrow \Gamma \gtrsim_I \Gamma''$

Proposition 6.

IREQ_REF $\quad \mathtt{ok}\ \Gamma \Rightarrow [\Gamma] \gtrsim_I [\Gamma]$

IREQ_ANTSYM $\quad [\Gamma] \gtrsim_I [\Gamma'] \wedge [\Gamma'] \gtrsim_I [\Gamma] \Rightarrow [\Gamma] = [\Gamma']$

IREQ_TRANS $\quad [\Gamma] \gtrsim_I [\Gamma'] \wedge [\Gamma'] \gtrsim_I [\Gamma''] \Rightarrow [\Gamma] \gtrsim_I [\Gamma'']$

Proposition 7.

EQ_IR_AN
$$(\Gamma_A : t_A) \gtrsim_I (\Gamma_N : t_N) \wedge$$
$$\forall \overline{x} \notin L \subseteq Id\,.\,\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{x} \mapsto \overline{s}_A{}^{\overline{x}}) : w_A{}^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}_A{}^{\overline{x}}) = \overline{s}_A \wedge {}^{\backslash \overline{x}}(w_A{}^{\overline{x}}) = w_A$$
$$\Rightarrow \exists \overline{y} \notin L\,.\,\exists \overline{s}_N \subset \mathit{LNExp}\,.\,\exists w_N \in \mathit{LNVal}\,.$$
$$\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}} \wedge {}^{\backslash \overline{z}}(\overline{s}_N{}^{\overline{z}}) = \overline{s}_N \wedge {}^{\backslash \overline{z}}(w_N{}^{\overline{z}}) = w_N \wedge \overline{z} \subseteq \overline{y}$$
$$\wedge ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$$

EQ_IR_NA
$$(\Gamma_A : t_A) \gtrsim_I (\Gamma_N : t_N) \wedge$$
$$\forall \overline{x} \notin L \subseteq Id\,.\,\Gamma_N : t_N \Downarrow^N (\Gamma_N, \overline{x} \mapsto \overline{s}_N{}^{\overline{x}}) : w_N{}^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}_N{}^{\overline{x}}) = \overline{s}_N \wedge {}^{\backslash \overline{x}}(w_N{}^{\overline{x}}) = w_N$$
$$\Rightarrow \exists \overline{z} \notin L\,.\,\exists \overline{s}_A \subset \mathit{LNExp}\,.\,\exists w_A \in \mathit{LNVal}\,.$$
$$\Gamma_A : t_A \Downarrow^A (\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}} \wedge {}^{\backslash \overline{y}}(\overline{s}_A{}^{\overline{y}}) = \overline{s}_A \wedge {}^{\backslash \overline{y}}(w_A{}^{\overline{y}}) = w_A \wedge \overline{z} \subseteq \overline{y}$$
$$\wedge ((\Gamma_A, \overline{y} \mapsto \overline{s}_A{}^{\overline{y}}) : w_A{}^{\overline{y}}) \gtrsim_I ((\Gamma_N, \overline{z} \mapsto \overline{s}_N{}^{\overline{z}}) : w_N{}^{\overline{z}})$$

Lemma 1.

SS_SUBST $\quad t[y/x] \sim_S t$

Lemma 2.

SS_OP $\quad |\overline{x}| = |\overline{y}| \Rightarrow t^{\overline{x}} \sim_S t^{\overline{y}}$

Lemma 3.

CLOSE_OPEN $\quad \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ t \Leftrightarrow {}^{\backslash \overline{x}}(t^{\overline{x}}) = t$

Lemma 4.

OPEN_CLOSE $\quad \mathtt{lc}\ t \Rightarrow ({}^{\backslash \overline{x}}t)^{\overline{x}} = t$

Lemma 5.

SS_LC $\quad t \sim_S t' \wedge \mathtt{lc}\ t \Rightarrow \mathtt{lc}\ t'$

Lemma 6.

LC_OP_VARS $\quad \mathtt{lc}\ t \wedge \overline{x} \subseteq Id \Rightarrow \exists s \in \mathit{LNExp}.\,(\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ s \wedge s^{\overline{x}} = t)$

Lemma 7.

REGULARITY $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \wedge \mathtt{ok}\ \Delta \wedge \mathtt{lc}\ w$

Lemma 8.

DEF_NOT_LOST $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta)$

Lemma 9.

NO_UPDATE $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \Gamma \subseteq \Delta$
where $\Downarrow^K$ represents $\Downarrow^N$ and $\Downarrow^A$

Lemma 10.

ADD_NAMES $\quad \Gamma : t \Downarrow^K \Delta : w \Rightarrow \mathtt{names}(\Delta : w) \subseteq \mathtt{names}(\Gamma : t) \cup \mathtt{dom}(\Delta)$

Lemma 11.

NEW_NAMES1 $\quad \Gamma : t \Downarrow^N \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ \Gamma$

NEW_NAMES2 $\quad \Gamma : t \Downarrow^A \Delta : w \wedge x \in \mathtt{dom}(\Delta) - \mathtt{dom}(\Gamma) \Rightarrow \mathtt{fresh}\ x\ \mathtt{in}\ (\Gamma : t)$

Lemma 12.

RENAMING1 $\quad \Gamma : t \Downarrow^K \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w)$
$\Rightarrow \Gamma[y/x] : t[y/x] \Downarrow^K \Delta[y/x] : w[y/x]$

RENAMING2 $\quad \Gamma : t \Downarrow^K \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w) \wedge x \notin \mathtt{dom}(\Gamma) \wedge x \in \mathtt{dom}(\Delta)$
$\Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$

Lemma 13.

CE_SS $\quad t \approx^V t' \Rightarrow t \sim_S t'$

Lemma 14.

CE_SUB $\quad t \approx^V t' \wedge V' \subseteq V \Rightarrow t \approx^{V'} t'$

CE_ADD $\quad t \approx^V t' \wedge \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ t \wedge \mathtt{fresh}\ \overline{x}\ \mathtt{in}\ t' \Rightarrow t \approx^{V \cup \overline{x}} t'$

Lemma 15.

CE_SUBS1 $\quad t \approx^V t' \wedge x, y \notin V \Rightarrow t[y/x] \approx^V t'$

CE_SUBS2 $\quad t \approx^V t' \wedge (\mathtt{fvar}\ y) \approx^V (\mathtt{fvar}\ y') \wedge x \in V \Rightarrow t[y/x] \approx^V t'[y'/x]$

CE_SUBS3 $\quad t \approx^V t' \wedge y \notin V \wedge \mathtt{fresh}\ y\ \mathtt{in}\ t \wedge \mathtt{fresh}\ y\ \mathtt{in}\ t' \Rightarrow t[y/x] \approx^{V[y/x]} t'[y/x]$

CE_OP1 $\quad t \approx^V t' \Rightarrow t^{\overline{x}} \approx^V t'^{\overline{x}}$

CE_OP2 $\quad t \approx^V t' \wedge \overline{x}, \overline{y} \notin V \wedge |\overline{x}| = |\overline{y}| \Rightarrow t^{\overline{x}} \approx^V t'^{\overline{y}}$

CE_OP3 $\quad t \approx^V t' \wedge (\mathtt{fvar}\ x) \approx^V (\mathtt{fvar}\ y) \Rightarrow t^x \approx^V t'^y$

Lemma 16.

HCE_DOM $\quad \Gamma \approx^V \Gamma' \Rightarrow \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma')$

HCE_IND $\quad \Gamma \approx^V \Gamma' \Rightarrow \mathtt{Ind}(\Gamma) = \mathtt{Ind}(\Gamma')$

HCE_OK $\quad \Gamma \approx^V \Gamma' \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma'$

Lemma 17.

HCE_ALT $\quad \Gamma \approx^V \Gamma' \Leftrightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma' \wedge \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \wedge (x \mapsto t \in \Gamma \wedge x \mapsto t' \in \Gamma' \Rightarrow t \approx^V t')$

Lemma 18.

HCE_SWAP $\quad \mathtt{ok}\ \Gamma \wedge x, y \in \mathtt{Ind}(\Gamma) \wedge x \neq y \Rightarrow \Gamma \ominus [x, y] \approx^{V - \{x,y\}} \Gamma \ominus [y, x]$

Lemma 19.

HE_PERM $\quad \mathtt{ok}\ \Gamma \wedge \overline{x}, \overline{y} \in \mathtt{Ind}(\Gamma) \Rightarrow (\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \Leftrightarrow \overline{y} \in \mathcal{S}(\overline{x}))$

Lemma 20.

IR_DOM $\quad \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{dom}(\Gamma') \subseteq \mathtt{dom}(\Gamma)$

IR_IND $\quad \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{Ind}(\Gamma') \subseteq \mathtt{Ind}(\Gamma)$

IR_OK $\quad \Gamma \succsim_I \Gamma' \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{ok}\ \Gamma'$

IR_DOM_HE $\quad \Gamma \succsim_I \Gamma' \wedge \mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma') \Rightarrow \Gamma \approx \Gamma'$

IR_IR_HE $\quad (\Gamma \succsim_I \Gamma' \wedge \Gamma' \succsim_I \Gamma) \Leftrightarrow \Gamma \approx \Gamma'$

Lemma 21.

IREQ_HE_IREQ1 $\quad [\Gamma] \succsim_I [\Gamma'] \wedge \Delta \approx \Gamma \Rightarrow [\Delta] \succsim_I [\Gamma']$

IREQ_HE_IREQ2 $\quad [\Gamma] \succsim_I [\Gamma'] \wedge \Delta \approx \Gamma' \Rightarrow [\Gamma] \succsim_I [\Delta]$

Lemma 22.

IRHT_IRH $\quad (\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow \Gamma \succsim_I \Gamma'$

IRHT_SS $\quad (\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow t \sim_S t'$

IRHT_LC $\quad (\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow \mathtt{lc}\ t \wedge \mathtt{lc}\ t'$

Lemma 23.
SS_OPK $\quad t \sim_S t' \wedge |\overline{x}| = |\overline{y}| \Rightarrow \{k \to \overline{x}\}t \sim_S \{k \to \overline{y}\}t'$

Lemma 24.
OPK_CLK_FRESH $\quad \{k \leftarrow \overline{x}\}(\{k \to \overline{x}\}t) = t \Rightarrow \texttt{fresh}\ \overline{x}\ \texttt{in}\ t$

Lemma 25.
LC_OPK_VARS $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \wedge \overline{x} \subseteq Id \Rightarrow \exists s \in \mathit{LNExp}.\, (\texttt{fresh}\ \overline{x}\ \texttt{in}\ s \wedge \{k \to \overline{x}\}s = t)$

Lemma 26.
KEEP_NAMES $\quad \Gamma : t \Downarrow^A \Delta : w \Rightarrow \texttt{fv}(t) \subseteq \texttt{names}(\Delta : w)$

Lemma 27.
CE_OPK1 $\quad t \approx^V t' \Rightarrow \{k \to \overline{x}\}t \approx^V \{k \to \overline{x}\}t'$
CE_OPK2 $\quad t \approx^V t' \wedge \overline{x}, \overline{y} \notin V \wedge |\overline{x}| = |\overline{y}| \Rightarrow \{k \to \overline{x}\}t \approx^V \{k \to \overline{y}\}t'$

Lemma 28.
HCE_BIND $\quad (\Gamma, x \mapsto t) \approx^V (\Gamma', x \mapsto t') \Rightarrow \Gamma \approx^V \Gamma' \wedge t \approx^V t'$

Lemma 29.
IND_DOM $\quad \texttt{ok}\ \Gamma \wedge x \in \texttt{Ind}(\Gamma) \Rightarrow \texttt{dom}(\Gamma \ominus x) = \texttt{dom}(\Gamma) - \{x\} \wedge \texttt{Ind}(\Gamma \ominus x) = \texttt{Ind}(\Gamma) - \{x\}$

Lemma 30.
IND_OK $\quad \texttt{ok}\ \Gamma \wedge x \in \texttt{Ind}(\Gamma) \Rightarrow \texttt{ok}\ (\Gamma \ominus x)$

Lemma 31.
IND_SUBS $\quad x \notin \texttt{dom}(\Gamma) \Rightarrow (\Gamma, x \mapsto \texttt{fvar}\ y) \ominus x = \Gamma[y/x]$

Lemma 32.
HCE_SUB $\quad \Gamma \approx^V \Gamma' \wedge V' \subseteq V \Rightarrow \Gamma \approx^{V'} \Gamma'$
HCE_ADD $\quad \Gamma \approx^V \Gamma' \wedge \overline{x} \notin \texttt{names}(\Gamma) \cup \texttt{names}(\Gamma') \Rightarrow \Gamma \approx^{V \cup \overline{x}} \Gamma'$

Lemma 33.
HCE_DEL_IND $\quad \Gamma \approx^V \Gamma' \wedge \texttt{dom}(\Gamma) \subseteq V \wedge x \in \texttt{Ind}(\Gamma) \Rightarrow \Gamma \ominus x \approx^V \Gamma' \ominus x$
HE_DEL_IND $\quad \Gamma \approx \Gamma' \wedge x \in \texttt{Ind}(\Gamma) \Rightarrow \Gamma \ominus x \approx \Gamma' \ominus x$

Lemma 34.
IR_FVAR $\quad (\Gamma : \texttt{fvar}\ x) \gtrsim_I (\Gamma' : \texttt{fvar}\ x')$
$\Rightarrow (x \notin \texttt{dom}(\Gamma)\ \wedge\ x' \notin \texttt{dom}(\Gamma'))$
$\vee\, (x \in \texttt{dom}(\Gamma)\ \wedge\ x' \in \texttt{dom}(\Gamma') \wedge$
$\exists\, x_0, \ldots, x_n \in Id\,.\, x_0 = x \wedge x_n = x'\ \wedge\ \forall i : 0 \le i < n\,.\, x_i \mapsto \texttt{fvar}\ x_{i+1} \in \Gamma)$

Lemma 35.
IR_IRHT $\quad (\Gamma, x \mapsto t) \gtrsim_I (\Gamma', x \mapsto t') \Rightarrow ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x \mapsto t') : t')$.

Lemma 36.
IR_FVAR_IRHT $\quad ((\Gamma, x \mapsto t) : \texttt{fvar}\ x) \gtrsim_I ((\Gamma', x' \mapsto t') : \texttt{fvar}\ x')$
$\Rightarrow ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x' \mapsto t') : t') \vee ((\Gamma, x \mapsto t) : t) \gtrsim_I ((\Gamma', x' \mapsto t') : \texttt{fvar}\ x')$.

Lemma 37.
IRHT_APP $\quad (\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x)) \gtrsim_I (\Gamma' : \texttt{app}\ t'\ (\texttt{fvar}\ x'))$
$\Rightarrow (\Gamma : t) \gtrsim_I (\Gamma' : t') \wedge (\Gamma : \texttt{fvar}\ x) \gtrsim_I (\Gamma' : \texttt{fvar}\ x')$.

Lemma 38.
IRHT_FV $(\Gamma : \texttt{fvar}\ x) \gtrsim_I (\Gamma' : \texttt{fvar}\ x') \wedge \Gamma \subseteq \Delta \wedge \Gamma' \subseteq \Delta' \wedge \Delta \gtrsim_I \Delta' \wedge$
$(x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin \texttt{dom}(\Delta)) \wedge (x' \notin \texttt{dom}(\Gamma') \Rightarrow x' \notin \texttt{dom}(\Delta')) \wedge$
$(y \in \texttt{dom}(\Gamma) \wedge y \notin \texttt{dom}(\Gamma') \Rightarrow y \notin \texttt{dom}(\Delta'))$
$\Rightarrow (\Delta : \texttt{fvar}\ x) \gtrsim_I (\Delta' : \texttt{fvar}\ x')$.

Lemma 39.
IRHT_RED_IND $\texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge y \neq x \wedge (\Gamma : \texttt{abs}\ t) \gtrsim_I (\Gamma' : \texttt{abs}\ t') \wedge (\Gamma : \texttt{fvar}\ x) \gtrsim_I (\Gamma' : \texttt{fvar}\ x')$
$\Rightarrow ((\Gamma, y \mapsto \texttt{fvar}\ x) : t^y) \gtrsim_I (\Gamma' : t'^{x'})$.

Lemma 40.
INTR_VARS $\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma' : \texttt{let}\ \overline{t'}\ \texttt{in}\ t') \wedge$
$(\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t) \gtrsim_I (\Gamma' : \texttt{let}\ \overline{t'}\ \texttt{in}\ t') \Rightarrow ((\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}}) \gtrsim_I ((\Gamma', \overline{x} \mapsto \overline{t'}^{\overline{x}}) : t'^{\overline{x}})$.

Lemma 41.
HCE_SUBS $\Gamma \approx^V \Gamma' \wedge y \notin V \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma' \Rightarrow \Gamma[y/x] \approx^{V[y/x]} \Gamma'[y/x]$
IR_SUBS $\Gamma \gtrsim_I \Gamma' \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma' \Rightarrow \Gamma[y/x] \gtrsim_I \Gamma'[y/x]$
IR_HT_SUBS $(\Gamma : t) \gtrsim_I (\Gamma' : t') \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma' : t')$
$\Rightarrow (\Gamma[y/x] : t[y/x]) \gtrsim_I (\Gamma'[y/x] : t'[y/x])$

Lemma 42.
IRHT_REF $\texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \Rightarrow (\Gamma : t) \gtrsim_I (\Gamma : t)$

Corollary 1.
IR_DOM_DOM $\Gamma \gtrsim_I \Gamma' \Rightarrow \Gamma \ominus (\texttt{dom}(\Gamma) - \texttt{dom}(\Gamma')) \approx \Gamma'$

Corollary 2.
RENAMING3 $\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Delta : w) \wedge x \notin \texttt{names}(\Gamma : t)$
$\Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$

Corollary 3.
CE_LC $t \approx^V t' \wedge \texttt{lc}\ t \Rightarrow \texttt{lc}\ t'$

Apéndice B

Trabajo en progreso

En este segundo apéndice se incluyen dos trabajos en progreso (WP1 [SGHHOM14a] y WP2 [SGHHOM12a]) que fueron presentados en sendas ediciones de las *Jornadas de Programación y Lenguajes* (PROLE) y publicados en las correspondientes actas. El primero de ellos, *Launchbury's semantics revisited: On the equivalence of context-heap semantics*, ha sido presentado en Cádiz en septiembre de 2014. En él se recopilan las principales ideas de las publicaciones P2 y P3 y se introduce el estudio que se está realizando actualmente sobre el efecto de actualizar o no las clausuras durante la evaluación. Se concluye con *A formalization in Coq of Launchbury's natural semantics for lazy evaluation*, presentado en Almería en septiembre de 2012. Este trabajo muestra los primeros pasos hacia la implementación en Coq de los resultados obtenidos en los trabajos sobre las semánticas.

Two works in progress, (WP1 [SGHHOM14a] *and* WP2 [SGHHOM12a])*, appear in this second appendix. They were presented in* Jornadas de Programación y Lenguajes *(PROLE) in 2014 and 2012, respectively. Both were published in the corresponding proceedings of the workshops. The first work,* Launchbury's semantics revisited: On the equivalence of context-heap semantics *was presented in Cadiz in September 2014, and collects the main ideas of publications* P2 *and* P3*. This paper also contains some of the results that we have already obtained on the effects of updating closures during evaluation. We conclude with* A formalization in Coq of Launchbury's natural semantics for lazy evaluation *presented in Almeria in September 2012. This work shows the first steps on the implementation in the proof assistant* Coq *of some of our results on semantics for lazy evaluation.*

Launchbury's semantics revisited: On the equivalence of context-heap semantics (Work in progress)

Lidia Sánchez-Gil
Facultad de Informática
Universidad Complutense de Madrid
España
isanche@ucm.es

Mercedes Hidalgo-Herrero
Facultad de Educación
Universidad Complutense de Madrid
España
mhidalgo@ucm.es

Yolanda Ortega-Mallén
Facultad de CC. Matemáticas
Universidad Complutense de Madrid
España
yolanda@ucm.es

Launchbury's natural semantics for lazy evaluation is based on heaps of bindings, i.e., variable-expression pairs, which define the evaluation context. In order to prove the adequacy of the operational semantics with respect to a standard denotational one, Lauchbury defines an alternative natural semantics where updating of bindings is removed and β-reduction is done through indirections instead of variable substitution. We study how context heaps are affected by these changes, and we define several relations between heaps. These relations allow to establish the equivalence between Launchbury's natural semantics and its alternative version. This result is crucial because many authors have based their proofs on its veracity.

1 Motivation

More than twenty years have elapsed since Launchbury first presented in [9] his natural semantics for lazy evaluation, a key contribution to the semantic foundation for non-strict functional programming languages like Haskell or Clean. Throughout these years, Launchbury's natural semantics has been cited frequently and has inspired many further works as well as several extensions like in [2, 10, 18, 8]. The authors have extended in [13] Launchbury's semantics with rules for *parallel application* that creates new processes to distribute the computation; these distributed processes exchange values through communication channels. The success of Lanchbury's proposal resides in its simplicity. Expressions are evaluated with respect to a *context*, which is represented by a heap of *bindings*, that is, (variable, expression) pairs. This heap is explicitly managed to make possible the sharing of bindings, thus, modeling laziness.

In order to prove that this lazy (operational) semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury introduces some variations in the operational semantics. On the one hand, the update of bindings with their computed values is an operational notion without counterpart in the standard denotational semantics, so that the alternative natural semantics does no longer update bindings and becomes a *call-by-name* semantics. On the other hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach to this mechanism, in the alternative operational semantics applications

Submitted to:
PROLE 2014

are carried out by introducing *indirections*, i.e., variables bound to variables, instead of by performing the β-reduction through substitution. Besides, the denotation "undefined" indicates that there is no value associated to the expression being evaluated, but there is no indication of the reason for that. By contrast, in the operational semantics there are two possibilities for not reaching a value: either the reduction gets blocked if no rule is applicable (*blackhole*), or the reduction never stops. The rules in the alternative semantics guarantee that reductions never reach a blackhole.

Unfortunately, the proof of the equivalence between the natural semantics and its alternative version is detailed nowhere, and a simple induction turns out to be insufficient. The *context-heap* semantics is too sensitive to the changes introduced by the alternative rules. Intuitively, both reduction systems should lead to the same results. However, this cannot be directly established since final values may contain free variables that are dependent on the context of evaluation, which is represented by the heap of bindings. The lack of update leads to the duplication of bindings, but is awkward to prove that duplicated bindings, as well as indirections, do not add relevant information to the context. Therefore, our challenge is to establish a way of relating the heaps and values obtained with each reduction system, and to prove that the semantics are equivalent, so that any reduction of a term in one of the systems has its counterpart in the other. To achieve this goal indirections and update are considered separately giving place to two intermediate semantics. We focus on the one with non-update. Our aim is to proof the equivalence between it and the two semantics proposed by Launchbury. The proof that deals with indirections will soon appear in [17] while the relation involving update is currently in progress.

We want to identify terms up to α-conversion, but dealing with α-equated terms usually implies the use of Barendregt's variable convention [3] to avoid the renaming of bound variables. However, the use of the variable convention is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [19]). Moreover, we intend to formalize our results with the help of some proof assistant like Coq [4] or Isabelle [11]. Looking for a binding system susceptible of formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [7]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [6], while free variables preserve their names. This is suitable in our case because context heaps collect free variables whose names we are interested in preserving in order to identify them more easily. A locally nameless version of Launchbury's natural semantics has been presented by the authors in [14] and [15].

Others are revisiting Launchbury's semantics too. For instance, Breitner has formally proven in [5] the correctness of the natural semantics by using Isabelle's nominal package [20], and presently he is working on the formalization of the adequacy. While Breitner is exclusively interested in formalizing the proofs, we have a broader objective: To analyze the effect of introducing indirections in the context heaps, and the correspondence between heap/value pairs obtained with update and those produced without update. Furthermore, we want to prove the equivalence of the two operational semantics.

The paper is structured as follows: In the next section we give an overview of the mentioned locally nameless version of Launchbury's natural semantics and its alternative rules. We define two intermediate semantics: one introducing indirections, and the other eliminating updates and blackholes. Section 3 is dedicated to indirections, while in Section 4 we study the similarities and differences between the reductions proofs obtained with and without update of bindings. In the last section we draw conclusions and outline our future work.

$$\begin{array}{lcll} x & \in & Var & \\ e & \in & Exp & ::= \quad x \mid \lambda x.e \mid (e\ x) \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \end{array}$$

Figure 1: Named representation of the extended λ-calculus

$$\begin{array}{lcl} x \in Id & & i, j \in \mathbb{N} \\ v \in Var & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\ t \in LNExp & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t \end{array}$$

Figure 2: Locally nameless syntax

2 A locally nameless representation

The language described in [9] is a lambda calculus extended with recursive local declarations. The abstract syntax, in the *named representation*, appears in Figure 1. Since there are two name binders, i.e., λ-abstraction and `let`-declaration, a quotient structure respect to α-equivalence is required. We avoid this by employing a *locally nameless representation* [7].

As mentioned above, our locally nameless representation has already been presented in [14] and [15]. Here we give only a brief presentation avoiding those technicalities that are not essential to the contributions of the present work.

2.1 Locally nameless syntax

The locally nameless version of the abstract syntax is shown in Figure 2. *Bound variables* and *free variables* are distinguished. Since `let`-declarations are multibinders, we have followed Charguéraud [7] and bound variables are represented with two natural numbers: the first number is a de Bruijn index that counts how many binders (abstraction or `let`) have been passed through in the syntactic tree to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind one variable, so that the second number should be zero.

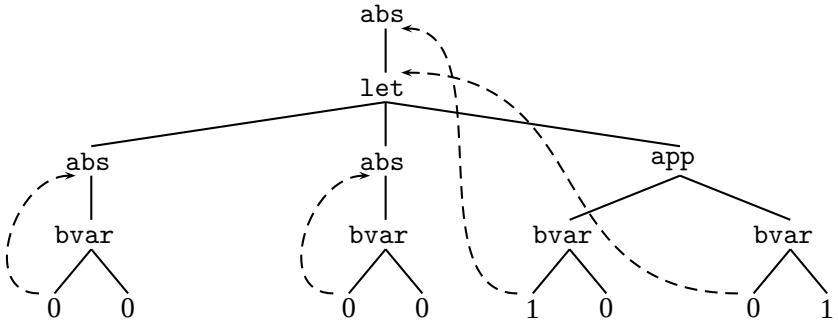
Example 1. *Let $e \in Exp$ be the λ-expression given in the named representation*

$$e \equiv \lambda z.\texttt{let}\ \{x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2\}\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in LNExp$ is:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \{\texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\}\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms. The syntactic tree is:



□

This locally nameless syntax allows to build terms that have no corresponding named expression in *Exp* (Figure 1). For instance, when bound variables indices are out of range. The terms in *LNExp* that do match expressions in *Exp* are called *locally-closed*, written $\texttt{lc}\ t$. The *local closure* predicate is detailed in [15]. We avoid those technicalities that are not essential to the new contributions of this work.

In the following, a list like $\{t_i\}_{i=1}^{n}$ is represented as $\overline{t}$, with length $|\overline{t}| = n$. Later on, we use the notation $[t : \overline{t}]$ to represent a list with head t and tail $\overline{t}$, and $+\!\!+$ for the concatenation of lists.

We denote by $\texttt{fv}(t)$ the set of *free variables* of a term t. A name $x \in Id$ is *fresh in a term* $t \in LNExp$, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t, i.e., $x \notin \texttt{fv}(t)$. Similarly, for a list of names, $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t$ if $\overline{x} \notin \texttt{fv}(t)$, where $\overline{x}$ represents a list of pairwise-distinct names in Id.

We say that two terms have the *same structure*, written $t \sim_S t'$, if they differ only in the names of their free variables.

Since there is no danger of name capture, *substitution* of variable names in a term is trivial in the locally nameless representation. We write $t[y/x]$ for replacing the occurrences of x by y in the term t. Clearly, name substitution preserves the structure of a term.

A *variable opening* operation is needed to manipulate locally nameless terms. This operation turns the outermost bound variables into free variables. The opening of a term $t \in LNExp$ with a list of names $\overline{x} \subseteq Id$ is denoted by $t^{\overline{x}}$. For simplicity, we write t^x for the variable opening with a unitary list $[x]$. A formal definition of variable opening can be found in [7] and [15]. Here we just illustrate the concept and its use with an example.

Example 2. *Let* $t \equiv \texttt{abs}\ (\texttt{let bvar 0 1}, \texttt{bvar 1 0 in app}\ (\texttt{abs bvar 2 0})\ (\texttt{bvar 0 1}))$. *Hence, the body of the abstraction is:*

$$u \equiv \texttt{let bvar 0 1}, \boxed{\texttt{bvar 1 0}}\ \texttt{in app}\ (\texttt{abs}\ \boxed{\texttt{bvar 2 0}})\ (\texttt{bvar 0 1}).$$

But then in u the bound variables referring to the outermost abstraction (shown squared) point to nowhere. Therefore, we consider u^x *instead of u, where*

$$u^x \quad = \quad \texttt{let bvar 0 1}, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar 0 1}).$$ □

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct).

Example 3. *We close the term obtained by opening u in Example 2.*
Let $t \equiv \texttt{let bvar 0 1}, \texttt{fvar}\ x\ \texttt{in app}\ (\texttt{abs fvar}\ x)\ (\texttt{bvar 0 1})$, *then*

$${}^{\backslash x}t \quad = \quad \texttt{let bvar 0 1}, \texttt{bvar 1 0 in app}\ (\texttt{abs bvar 2 0})\ (\texttt{bvar 0 1}).$$ □

Notice that in the last example the closed term coincides with u, the body of the abstraction in Example 2 that was opened with x, although this is not always the case. Only under some conditions variable closing and variable opening are inverse operations: If the variables are fresh in t, then ${}^{\backslash\overline{x}}(t^{\overline{x}}) = t$, and if the term is locally closed, then $({}^{\backslash\overline{x}}t)^{\overline{x}} = t$.

$$\textsc{LNLam}\quad \frac{\{\mathtt{ok}\ \Gamma\} \qquad \{\mathtt{lc}\ (\mathtt{abs}\ t)\}}{\Gamma : \mathtt{abs}\ t \Downarrow \Gamma : \mathtt{abs}\ t}$$

$$\textsc{LNVar}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : \mathtt{fvar}\ x \Downarrow (\Delta, x \mapsto w) : w}$$

$$\textsc{LNApp}\quad \frac{\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \qquad \Theta : u^x \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin \mathtt{dom}(\Delta)\}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow \Delta : w}$$

$$\textsc{LNLet}\quad \frac{\begin{array}{c}\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\,.\,[(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}}) : w^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}^{\overline{x}}) = \overline{s} \wedge {}^{\backslash \overline{x}}(w^{\overline{x}}) = w] \\ \{\overline{y}^{|\overline{t}|} \notin L\}\end{array}}{\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{y}}) : w^{\overline{y}}}$$

Figure 3: Natural semantics with locally nameless representation

$$\textsc{ALNVar}\quad \frac{(\Gamma, x \mapsto t) : t \Downarrow \Delta : w}{(\Gamma, x \mapsto t) : \mathtt{fvar}\ x \Downarrow \Delta : w}$$

$$\textsc{ALNApp}\quad \frac{\begin{array}{c}\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \\ \forall y \notin L \subseteq Id\,.\,[(\Theta, y \mapsto \mathtt{fvar}\ x) : u^y \Downarrow ([y : \overline{z}] \mapsto \overline{s}^y) : w^y \wedge {}^{\backslash y}(\overline{s}^y) = \overline{s} \wedge {}^{\backslash y}(w^y) = w] \\ \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin [z : \overline{z}]\} \qquad \{z \notin L\}\end{array}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow ([z : \overline{z}] \mapsto \overline{s}^z) : w^z}$$

Figure 4: Alternative rules with locally nameless representation

2.2 Locally nameless semantics

In the natural semantics defined by Launchbury [9] judgements are of the form $\Gamma : t \Downarrow \Delta : w$, that is, the term t in the context of the heap Γ reduces to the value w in the context of the (modified) heap Δ. *Values* ($w \in Val$) are terms in weak-head-normal-form (*whnf*) and *heaps* are collections of *bindings*, i.e., pairs (variable, term). A binding $(\mathtt{fvar}\ x, t)$ with $x \in Id$ and $t \in LNExp$ is represented by $x \mapsto t$. In the following, we represent a heap $\{x_i \mapsto t_i\}_{i=1}^n$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\mathtt{dom}(\Gamma)$, collects the names that are defined in the heap, so that $\mathtt{dom}(\overline{x} \mapsto \overline{t}) = \overline{x}$. By contrast, the function `names` returns the set of all the names that appear in a heap, i.e., the names occurring either in the domain or in the terms in the right-hand side of the bindings. This is used to define a freshness predicate for heaps: $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ \Gamma = \overline{x} \notin \mathtt{names}(\Gamma)$.

In a well-formed heap names are defined at most once and terms are locally closed. We write $\mathtt{ok}\ \Gamma$ to indicate that a heap is well-formed.

In Figure 3 we show a locally nameless representation of the rules for the natural semantics for lazy evaluation, given by Launchbury in [9]. For clarity, in the rules we put in braces the side-conditions to better distinguish them from the judgements.

To prove the computational adequacy of the natural semantics (Figure 3) with respect to a standard denotational semantics, Launchbury introduces alternative rules for variables and applications, whose locally nameless version is shown in Figure 4. Observe that the ALNVar rule does not longer update the binding for the variable being evaluated, namely x. Besides, the binding for x does not disappear from the heap where the term bound to x is to be evaluated; therefore, any further reference to x leads to an infinite reduction. The effect of ALNApp is the addition of an indirection $y \mapsto \mathtt{fvar}\ x$ instead of

	NS	**INS**	**NNS**	**ANS**
Indirections	✗	✓	✗	✓
Update	✓	✓	✗	✗
Blackholes	✓	✓	✗	✗

Figure 5: The lazy natural semantics and its alternatives

performing the β-reduction by substitution, as in u^x in LNAPP.

In the rules LNLET and ALNAPP we use *cofinite quantification* [1], which is an alternative to "exists-fresh" quantifications that provides stronger induction and inversion principles. Although there are not explicit freshness side-conditions in the rules, the finite set L represents somehow the names that should be avoided during a reduction proof. We use the variable opening to express that the final heap and value may depend on the chosen names. For instance, in LNLET, $w^{\overline{x}}$ indicates that the final value depends on the names $\overline{x}$, but there is a common basis w. Moreover, it is required that this basis does not contain occurrences of $\overline{x}$; this is is expressed by ${}^{\backslash\overline{x}}(w^{\overline{x}}) = w$. A detailed explanation of these semantic rules can be found in [14, 15].

In the following, the natural semantics (rules in Figure 3) is referred as NS, and the alternative semantics (rules LNLAM, LNLET and those in Figure 4) as ANS. We write $\Downarrow^A$ for reductions in ANS. Launchbury proves in [9] the correctness of NS with respect to a standard denotational semantics, and a similar result for ANS is easily obtained (as the authors of this paper have done in [12]). Therefore, NS and ANS are "denotationally" equivalent in the sense that if an expression is reducible (in some heap context) by both semantics then the obtained values have the same denotation. But this is insufficient for our purposes, because we want to ensure that if for some (heap : term) pair a reduction exists in any of the semantics, then there must exist a reduction in the other too, and the final heaps must be related. The changes introduced by ANS might seem to involve no serious difficulties to prove the latter result. Unfortunately things are not so easy. On the one hand, the alternative rule for variables transforms the original call-by-need semantics into a call-by-name semantics because bindings are not updated and computed values are no longer shared. Moreover, in the original semantics the reduction of a self-reference gets blocked (*blackhole*), while in the alternative semantics self-references yield infinite reductions. On the other hand, the addition of indirections complicates the task of comparing the (heap : value) pairs obtained by each reduction system, as one may need to follow a chain of indirections to get the term bound to a variable. We deal separately with each modification and introduce two intermediate semantics: (1) the *No-update Natural Semantics* (NNS) with the rules of NS (Figure 3) except for the variable rule, that corresponds to the one in the alternative version, i.e., ALNVAR in Figure 4; and (2) the *Indirection Natural Semantics* (INS) with the rules of NS but for the application rule, that corresponds to the alternative ALNAPP rule in Figure 4. We use $\Downarrow^N$ to represent reductions of NNS and $\Downarrow^I$ for those of INS. Figure 5 resumes the characteristics of the four natural semantics explained above.

It is guaranteed that the judgements produced by the locally nameless rules given in Figures 3 and 4 involve only well-formed heaps and locally closed terms. Furthermore, the reduction systems corresponding to these rules verify a number of interesting properties proved in [15]. We just show here the *renaming* lemma, that ensures that the evaluation of a term is independent of the names chosen during the reduction process. Further, any name defined in the context heap can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reductions for (heap : term) pairs are unique up to α-conversion of the names defined in the heap.

Lemma 1. *(Renaming)*

1. $\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\, y \,\texttt{in}\, \Gamma,\Delta,t,w \Rightarrow \Gamma[y/x] : t[y/x] \Downarrow^K \Delta[y/x] : w[y/x]$;
2. $\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\, y \,\texttt{in}\, \Gamma,\Delta,t,w \wedge x \notin \texttt{dom}(\Gamma) \wedge x \in \texttt{dom}(\Delta) \Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$,

where $\Gamma[y/x]$ indicates that name substitution is done in the left and right hand sides of the heap Γ, and $\Downarrow^K$ represents $\Downarrow$, $\Downarrow^A$, $\Downarrow^I$ and $\Downarrow^N$.

3 Indirections

The aim in this section is to prove the equivalence of NNS and ANS. After the evaluation of a term in a given context, each semantics yields a different binding heap. It is necessary to analyze their differences, which lie in the indirections introduced by ANS. An *indirection* is a binding of the form $x \mapsto \texttt{fvar}\, y$, that is, it just redirects to another variable name. The set of indirections of a heap Γ is denoted by $\texttt{Ind}(\Gamma)$.

The next example illustrates the situation.

Example 4. *Let us evaluate the term*

$$t \equiv \quad \texttt{let abs (bvar 0 0) in app (abs } s\texttt{) (bvar 0 0)},$$

where

$$s \equiv \quad \texttt{let abs (bvar 0 0), app (bvar 0 0) (bvar 1 0) in abs (bvar 0 0)}$$

in the empty context $\Gamma = \emptyset$.

$$\begin{array}{lll} \Gamma : t & \Downarrow^N & \{x_0 \mapsto \texttt{abs (bvar 0 0)}, x_1 \mapsto \texttt{abs (bvar 0 0)}, x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } x_0\texttt{)}\} \\ & & : \texttt{abs (bvar 0 0)} \\ \Gamma : t & \Downarrow^A & \{x_0 \mapsto \texttt{abs (bvar 0 0)}, x_1 \mapsto \texttt{abs (bvar 0 0)}, x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } y\texttt{)}, y \mapsto \texttt{(fvar } x_0\texttt{)}\} \\ & & : \texttt{abs (bvar 0 0)} \end{array}$$

The value produced is the same in both cases. Yet, when comparing the final heap in $\Downarrow^A$ *with the final heap in* $\Downarrow^N$*, we observe that there is an extra indirection,* $y \mapsto \texttt{fvar}\, x_0$*. This indirection corresponds to the binding introduced by* ALNAPP *to reduce the application in the term* t*.* □

The previous example gives a hint of how to establish a relation between the heaps that are obtained with NNS and those produced by ANS: Two heaps are related if one can be obtained from the other by eliminating some indirections. For this purpose we define how to remove indirections from a heap, while preserving the evaluation context represented by that heap.

$$\begin{array}{rcl} (\emptyset, x \mapsto \texttt{fvar}\, y) \ominus x & = & \emptyset \\ ((\Gamma, z \mapsto t), x \mapsto \texttt{fvar}\, y) \ominus x & = & ((\Gamma, x \mapsto \texttt{fvar}\, y) \ominus x, z \mapsto t[y/x]) \end{array}$$

This definition can be generalized to remove a sequence of indirections from a heap:

$$\Gamma \ominus [\,] \;=\; \Gamma \qquad\qquad \Gamma \ominus [x : \overline{x}] \;=\; (\Gamma \ominus x) \ominus \overline{x}$$

3.1 Context equivalence

The meaning of a term depends on the meaning of its free variables. However, if a free variable is not defined in the context of evaluation of a term, then the name of this free variable is irrelevant. Therefore, we consider that two terms are equivalent in a given context if they only differ in the names of the free variables that do not belong to the context.

Definition 1. *Let $V \subseteq Id$, and $t,t' \in LNExp$. We say that t and t' are* context-equivalent in V*, written $t \approx^V t'$, when*

$$\text{CE-BVAR}\quad \frac{}{(\texttt{bvar}\ i\ j) \approx^V (\texttt{bvar}\ i\ j)} \qquad \text{CE-FVAR}\quad \frac{x,x' \notin V \vee x = x'}{(\texttt{fvar}\ x) \approx^V (\texttt{fvar}\ x')}$$

$$\text{CE-ABS}\quad \frac{t \approx^V t'}{(\texttt{abs}\ t) \approx^V (\texttt{abs}\ t')} \qquad \text{CE-APP}\quad \frac{t \approx^V t' \quad v \approx^V v'}{(\texttt{app}\ t\ v) \approx^V (\texttt{app}\ t'\ v')}$$

$$\text{CE-LET}\quad \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \approx^V \overline{t'} \quad t \approx^V t'}{(\texttt{let}\ \overline{t}\ \texttt{in}\ t) \approx^V (\texttt{let}\ \overline{t'}\ \texttt{in}\ t')}$$

Fixed the set of names V, $\approx^V$ is an equivalence relation on *LNExp*. Based on this equivalence on terms, we define a family of equivalences on heaps, where two heaps are considered equivalent when they have the same domain and the corresponding closures may differ only in the free variables not defined in a given context:

Definition 2. *Let $V \subseteq Id$, and $\Gamma,\Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-context-equivalent in V*, written $\Gamma \approx^V \Gamma'$, when*

$$\text{HCE-EMPTY}\quad \frac{}{\emptyset \approx^V \emptyset} \qquad \text{HCE-CONS}\quad \frac{\Gamma \approx^V \Gamma' \quad t \approx^V t' \quad \texttt{lc}\ t \quad x \notin \texttt{dom}(\Gamma)}{(\Gamma, x \mapsto t) \approx^V (\Gamma', x \mapsto t')}$$

There is an alternative characterization for heap-context-equivalence which expresses that two heaps are context-equivalent whenever they are well-formed, have the same domain, and each pair of corresponding bound terms is context-equivalent.

Lemma 2. $\Gamma \approx^V \Gamma' \Leftrightarrow \texttt{ok}\ \Gamma \wedge \texttt{ok}\ \Gamma' \wedge \texttt{dom}(\Gamma) = \texttt{dom}(\Gamma') \wedge (x \mapsto t \in \Gamma \wedge x \mapsto t' \in \Gamma' \Rightarrow t \approx^V t')$.

Considering context-equivalence on heaps, we are particularly interested in the case where the context coincides with the domain of the heaps:

Definition 3. *Let $\Gamma,\Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-equivalent*, written $\Gamma \approx \Gamma'$, if they are heap-context-equivalent in* $\texttt{dom}(\Gamma)$*, i.e., $\Gamma \approx^{\texttt{dom}(\Gamma)} \Gamma'$.*

If equivalent heaps are obtained by removing different sequences of indirections, then these must be the same up to permutation:

Lemma 3. $\texttt{ok}\ \Gamma \wedge \overline{x},\overline{y} \subseteq \texttt{Ind}(\Gamma) \Rightarrow (\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \Leftrightarrow \overline{y} \in \mathscr{S}(\overline{x}))$,
where $\mathscr{S}(\overline{x})$ denotes the set of all permutations of $\overline{x}$.

3.2 Indirection relation

Coming back to the idea of Example 4, where a heap can be obtained from another by just removing some indirections, we define the following relation on heaps:

Definition 4. *Let $\Gamma,\Gamma' \in LNHeap$. We say that Γ is* indirection-related *to Γ', written $\Gamma \gtrsim_I \Gamma'$, when*

$$\text{IR-HE}\quad \frac{\Gamma \approx \Gamma'}{\Gamma \gtrsim_I \Gamma'} \qquad \text{IR-IR}\quad \frac{\texttt{ok}\ \Gamma \quad \Gamma \ominus x \gtrsim_I \Gamma' \quad x \in \texttt{Ind}(\Gamma)}{\Gamma \gtrsim_I \Gamma'}$$

There is an alternative characterization for the relation $\gtrsim_I$ which expresses that a heap is indirection-related to another whenever the later can be obtained from the former by removing a sequence of indirections.

Proposition 1. $\Gamma \succsim_I \Gamma' \Leftrightarrow \mathtt{ok}\ \Gamma \wedge \exists\, \overline{x} \subseteq \mathtt{Ind}(\Gamma)\,.\,(\Gamma \ominus \overline{x}) \approx \Gamma'$.

By Lemma 3, the sequence of indirections is unique up to permutations, and it corresponds to the difference between the domains of the related heaps:

Corollary 1. $\Gamma \succsim_I \Gamma' \Rightarrow (\Gamma \ominus (\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma'))) \approx \Gamma'$. [1]

The *indirection-relation* is a preorder on the set of well-formed heaps. We extended the relation to (heap : term) pairs:

Definition 5. *Let* $\Gamma, \Gamma' \in$ *LNHeap, and* $t, t' \in$ *LNExp. We say that* $(\Gamma : t)$ *is* indirection-related *to* $(\Gamma' : t')$*, written* $(\Gamma : t) \succsim_I (\Gamma' : t')$*, if*

$$\text{IR-HT} \quad \frac{\forall z \notin L \subseteq Id\,.\,(\Gamma, z \mapsto t) \succsim_I (\Gamma', z \mapsto t')}{(\Gamma : t) \succsim_I (\Gamma' : t')}$$

We use cofinite quantification instead of adding freshness conditions on the new name z.

It is easy to prove that two (heap : term) pairs are indirection-related only if the heaps are indirection related and the terms have the same structure:

Lemma 4. $(\Gamma : t) \succsim_I (\Gamma' : t') \Rightarrow \Gamma \succsim_I \Gamma' \wedge t \sim_S t'$.

We illustrate these definitions with an example.

Example 5. *Let us consider the following heap and term:*

$$\begin{array}{rcl} \Gamma & = & \{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0)), \\ & & \quad y_0 \mapsto \mathtt{fvar}\ x_2\} \\ t & = & \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ \mathtt{bvar}\ 0\ 0) \end{array}$$

The (heap : term) pairs related with $(\Gamma : t)$ *are obtained by removing the sequences of indirections* $[\,]$*,* $[y_0]$*,* $[x_0]$*, and* $[x_0, y_0]$*:*

a) $\{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0)), y_0 \mapsto \mathtt{fvar}\ x_2\}$
$: \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0))$

b) $\{x_0 \mapsto \mathtt{fvar}\ x_1, x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0))\}$
$: \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{bvar}\ 0\ 0))$

c) $\{x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0)), y_0 \mapsto \mathtt{fvar}\ x_2\}$
$: \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0))$

d) $\{x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0))\}$
$: \mathtt{abs}\ (\mathtt{app}\ (\mathtt{fvar}\ x_1)\ (\mathtt{bvar}\ 0\ 0))$

□

Now we are ready to establish the equivalence between ANS and NNS in the sense that if a reduction proof can be obtained with ANS for some term in a given context heap, then there must exist a reduction proof in NNS for the same (heap : term) pair such that the final (heap : value) is indirection-related to the final (heap : value) obtained with ANS, and vice versa.

Theorem 1. *(Equivalence ANS-NNS).*

1. $\Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in$ *LNHeap*$\,.\,\exists w_N \in$ *Val*$\,.\,\Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \succsim_I (\Delta_N : w_N)$.

[1] Since the ordering of indirections is irrelevant, $\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$ represents any sequence with the names defined in Γ but undefined in Γ'.

2. $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in \mathit{LNHeap} . \exists w_A \in \mathit{Val} . \exists \overline{x} \subseteq \mathtt{dom}(\Delta_N) - \mathtt{dom}(\Gamma) . \exists \overline{y} \subseteq \mathit{Id} .$
$|\overline{x}| = |\overline{y}| \wedge \Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \succsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$.

Notice that in the second part of the theorem, i.e., from NNS to ANS, a renaming may be needed. This renaming only affects the names that are added to the heap during the reduction process. This is due to the fact that in NNS names occurring in the evaluation term (that is t in the theorem) may disappear during the evaluation and, consequently, they may be chosen on some application of the rule LNLET and added to the final heap. This cannot happen in ANS due to the alternative application rule.

The proof of Theorem 1 is not straightforward and induction cannot be applied directly. Several intermediate results are needed to prove a generalization of the theorem where instead of evaluating the same term in the same initial context heap, indirection-related initial (heap : term) pairs are considered. This is developed in detail in [16], and a reduced version will soon appear in [17].

4 No update

In this section we compare (heap : term) pairs obtained with NS, where bindings are updated with the values obtained during reduction, with those obtained with NNS, without update, and where infinite reductions may occur due to self-references. We start with an example.

Example 6. *Let us consider the following term:*

$$
\begin{array}{rcll}
t & \equiv & \mathtt{let} & \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \\
& & & \mathtt{let} \quad \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1) \\
& & & \mathtt{in} \quad \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1), \\
& & & \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1) \\
& & \mathtt{in} & \mathtt{app}\ (\mathtt{app}\ (\mathtt{bvar}\ 0\ 1)\ (\mathtt{bvar}\ 0\ 0))\ (\mathtt{bvar}\ 0\ 2)
\end{array}
$$

When the term t is evaluated in the context of the empty heap the following final (heap : value) pairs are obtained:

$$
\begin{array}{ll}
\Downarrow & \{x_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \\
& \ y_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ y_0)\ (\mathtt{fvar}\ y_1)\} \\
& : \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0)
\end{array}
$$

$$
\begin{array}{ll}
\Downarrow^N & \{x_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \\
& \ x_1 \mapsto \ \mathtt{let} \quad \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1) \\
& \qquad \mathtt{in} \quad \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1), \\
& \ x_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{fvar}\ x_1), \\
& \ y_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ y_0)\ (\mathtt{fvar}\ y_1), \\
& \ y'_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y'_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y'_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ y'_0)\ (\mathtt{fvar}\ y'_1)\} \\
& : \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0)
\end{array}
$$

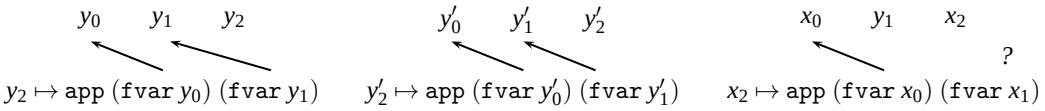
The inner `let`*-declaration, which is bound to the name x_1, is required twice. In the case of NS ($\Downarrow$), the evaluation of x_1 entails the introduction of three new names (y_0,y_1 and y_2), and the binding for x_1 is updated with the value obtained for the body term. Thus, the second time x_1 is required it is not re-evaluated. This is not the case in NNS ($\Downarrow^N$), where a second evaluation of x_1 implies the introduction of duplicated names (y'_0,y'_1 and y'_2) in the heap.* □

In the previous example one can observe the two main differences between the final heaps obtained by evaluating an expression with NS and with NNS. On the one hand, some variables that are bound to *whnf* values in NS remain bound to their initial terms in NNS. On the other hand, when evaluating with NNS, more bindings are obtained with respect to NS. The "extra" bindings are produced by duplicated evaluations of `let`-declarations. Therefore, to relate the final heaps we proceed in two steps: First we remove the extra bindings of the final heap of NNS to obtain a heap with the same domain as the one obtained with NS; second we check that the bindings that have not been updated are "equivalent" to the corresponding updated bindings.

4.1 Group relation

The first step is to identify duplicated bindings, i.e., those that correspond to the re-evaluation of a `let`-declaration. The next example illustrates the problem.

Example 7. *We choose three groups of bindings from the heap obtained with the $\Downarrow^N$-reduction in Example 6: $\overline{y} = [y_0, y_1, y_2]$, $\overline{y'} = [y'_0, y'_1, y'_2]$ and $\overline{z} = [x_0, y_1, x_2]$.*



We observe that y_0, y'_0 and x_0 are bound to terms with the same structure. *Similarly for y_1, y'_1 and y_1, and for y_2, y'_2 and x_2. But a closer look detects that $[x_0, y_1, x_2]$ is different from the other two groups: If the terms bound in each group are closed with the names of that group, then equal terms are obtained in the first two groups, while a different term is obtained in the third group:*

$$
\begin{array}{llllllllll}
y_0 & \xrightarrow{\backslash \overline{y}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y_1 & \xrightarrow{\backslash \overline{y}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y_2 & \xrightarrow{\backslash \overline{y}} & \texttt{app}\ (\texttt{bvar}\ 0\ 0)\ (\texttt{bvar}\ 0\ 1) \\
y'_0 & \xrightarrow{\backslash \overline{y'}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y'_1 & \xrightarrow{\backslash \overline{y'}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y'_2 & \xrightarrow{\backslash \overline{y'}} & \texttt{app}\ (\texttt{bvar}\ 0\ 0)\ (\texttt{bvar}\ 0\ 1) \\
x_0 & \xrightarrow{\backslash \overline{z}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y_1 & \xrightarrow{\backslash \overline{z}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & x_2 & \xrightarrow{\backslash \overline{z}} & \texttt{app}\ (\texttt{bvar}\ 0\ 0)\ (\texttt{fvar}\ x_1)
\end{array}
$$

Therefore, groups $\overline{y}$ and $\overline{y'}$ should be related, but not with $\overline{z}$. □

We start by relating terms (with respect to two lists of names) that are equal except for the free variables, and those names that are different occupy the same position in their respective lists.

Definition 6. *Let $t, t' \in LNExp$ and $\overline{x}, \overline{y} \subseteq Id$. We say that t and t' are* context-group-related *in the contexts of $\overline{x}$ and $\overline{y}$, written $t \approx^{(\overline{x},\overline{y})} t'$, when:*

$$\text{CR-BVAR}\quad \frac{|\overline{x}| = |\overline{y}|}{(\texttt{bvar}\ i\ j) \approx^{(\overline{x},\overline{y})} (\texttt{bvar}\ i\ j)} \qquad \text{CR-ABS}\quad \frac{t \approx^{(\overline{x},\overline{y})} t'}{(\texttt{abs}\ t) \approx^{(\overline{x},\overline{y})} (\texttt{abs}\ t')}$$

$$\text{CR-FVAR1}\quad \frac{|\overline{x}| = |\overline{y}| \quad x \notin \overline{x} \cup \overline{y}}{(\texttt{fvar}\ x) \approx^{(\overline{x},\overline{y})} (\texttt{fvar}\ x)} \qquad \text{CR-APP}\quad \frac{t \approx^{(\overline{x},\overline{y})} t' \quad v \approx^{(\overline{x},\overline{y})} v'}{(\texttt{app}\ t\ v) \approx^{(\overline{x},\overline{y})} (\texttt{app}\ t'\ v')}$$

$$\text{CR-FVAR2}\quad \frac{\begin{array}{c}|\overline{x}| = |\overline{y}| \\ x = \texttt{List.nth}\ i\ \overline{x} \quad y = \texttt{List.nth}\ i\ \overline{y}\end{array}}{(\texttt{fvar}\ x) \approx^{(\overline{x},\overline{y})} (\texttt{fvar}\ y)} \qquad \text{CR-LET}\quad \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \approx^{(\overline{x},\overline{y})} \overline{t'} \quad t \approx^{(\overline{x},\overline{y})} t'}{(\texttt{let}\ \overline{t}\ \texttt{in}\ t) \approx^{(\overline{x},\overline{y})} (\texttt{let}\ \overline{t'}\ \texttt{in}\ t')}$$

An alternative to the definition above is to check that the terms are equal under closure in their respective contexts:

Lemma 5. $t \approx^{(\overline{x},\overline{y})} t' \Leftrightarrow [t \sim_S t' \wedge {}^{\backslash \overline{x}}t = {}^{\backslash \overline{y}}t' \wedge |\overline{x}| = |\overline{y}|]$

Next we relate heaps that differ in duplicated groups of bindings.

Definition 7. *Let* $\Gamma, \Gamma' \in$ *LNHeap. We say that* Γ *is* group-related *to* Γ'*, written* $\Gamma \succsim_G \Gamma'$*, when*

$$\text{GR-EQ}\quad \frac{}{\Gamma \succsim_G \Gamma} \qquad \text{GR-GR}\quad \frac{\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s} \qquad \overline{x} \cap \overline{y} = \emptyset \qquad (\Gamma, \overline{x} \mapsto \overline{t})[\overline{x}/\overline{y}] \succsim_G \Gamma'}{(\Gamma, \overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{s}) \succsim_G \Gamma'}$$

This relation is a partial order on heaps. We extend it to (heap : term) pairs:

Definition 8. *Let* $\Gamma, \Gamma' \in$ *LNHeap, and* $t, t' \in$ *LNExp. We say that* $(\Gamma : t)$ *is* group-related *to* $(\Gamma' : t')$*, written* $(\Gamma : t) \succsim_G (\Gamma' : t')$*, when*

$$\text{GR\_HT\_EQ}\quad \frac{}{(\Gamma : t) \succsim_G (\Gamma : t)}$$

$$\text{GR\_HT\_GR}\quad \frac{\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s} \qquad \overline{x} \cap \overline{y} = \emptyset \qquad ((\Gamma, \overline{x} \mapsto \overline{t})[\overline{x}/\overline{y}] : t[\overline{x}/\overline{y}]) \succsim_G (\Gamma' : t')}{((\Gamma, \overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{s}) : t) \succsim_G (\Gamma' : t')}$$

Group-related (heap : term) pairs are equivalent in the sense that if there exists a NNS-reduction for one of them, then there also exists a NNS-reduction for the other, so that the final (heap : value) pairs are group-related too.

Lemma 6. $(\Gamma : t) \succsim_G (\Gamma' : t') \wedge \Gamma' : t' \Downarrow^N \Delta' : w' \Rightarrow$
$\exists \Delta \in$ *LNHeap*, $w \in$ *Val*$\,.\, \Gamma : t \Downarrow^N \Delta : w \wedge (\Delta : w) \succsim_G (\Delta' : w')$.

4.2 Update relation

Once all the duplicated groups of names have been detected and eliminated, we have to deal with updating. For this, we check that those bindings in the no-updated heap, which have unevaluated expressions do evaluate to values "equivalent" to those in the updated heap. For a recursive definition, we fix an initial context heap for these evaluations.

Definition 9. *Let* $\Gamma, \Gamma', \Delta \in$ *LNHeap. We say that* Γ *is* update-related *to* Γ' in the context of Δ*, written* $\Gamma \sim_U^\Delta \Gamma'$*, when*

$$\text{UCR\_EQ}\quad \frac{}{\Gamma \sim_U^\Delta \Gamma} \qquad \text{UCR\_VT}\quad \frac{\Gamma \sim_U^\Delta \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma, x \mapsto t) \sim_U^\Delta (\Gamma', x \mapsto w')}$$

Notice that, by definition, update related heaps have the same domain. We are particularly interested in the case where the context coincides with the first heap:

Definition 10. *Let* $\Gamma, \Gamma' \in$ *LNHeap. We say that* Γ *is* update-related *to* Γ'*, written* $\Gamma \sim_U \Gamma'$*, if* $\Gamma \sim_U^\Gamma \Gamma'$.

Once again we extend these definitions to (heap : term) pairs:

Definition 11. *Let* $\Gamma, \Gamma', \Delta \in$ *LNHeap, and* $t, t' \in$ *LNExp. We say that* $(\Gamma : t)$ *is* update-related *to* $(\Gamma' : t')$ in the context of Δ*, written* $(\Gamma : t) \sim_U^\Delta (\Gamma : t')$*, when*

$$\text{UCR\_TT\_HT}\quad \frac{\Gamma \sim_U^\Delta \Gamma'}{(\Gamma : t) \sim_U^\Delta (\Gamma' : t)}$$

$$\text{UCR\_VT\_HT}\quad \frac{\Gamma \sim_U^\Delta \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma : t) \sim_U^\Delta (\Gamma' : w')}$$

And $(\Gamma : t)$ *is* update-related *to* $(\Gamma' : t')$*, written* $(\Gamma : t) \sim_U (\Gamma' : t')$*, if* $(\Gamma : t) \sim_U^\Gamma (\Gamma' : t')$.

4.3 Group-update relation

Finally, we combine the group and the update relations to obtain the desired equivalence between heaps in NS and those in NNS.

Definition 12. *Let* $\Gamma, \Gamma' \in LNHeap$. *We say that* Γ *is* group-update-related *to* Γ', *written* $\Gamma \succsim_{GU} \Gamma'$, *when*

$$\text{GUR} \quad \frac{\Gamma \succsim_G \Delta \quad \Delta \sim_U \Gamma' \quad \mathtt{ok}\,\Gamma \quad \mathtt{ok}\,\Gamma'}{\Gamma \succsim_{GU} \Gamma'}$$

And the extension to (heap : term) pairs:

$$\text{GUR\_HT} \quad \frac{(\Gamma : t) \succsim_G (\Delta : s) \quad (\Delta : s) \sim_U (\Gamma' : t') \quad \mathtt{ok}\,\Gamma \quad \mathtt{ok}\,\Gamma' \quad \mathtt{lc}\,t \quad \mathtt{lc}\,t'}{(\Gamma : t) \succsim_{GU} (\Gamma' : t')}$$

We define the equivalence between NS and NNS in similar fashion to the equivalence ANS-NNS (Theorem 1), that is, if a reduction proof can be obtained with NS for some term in a given context heap, then there must exist a reduction proof in NNS for that same (heap : term) pair such that the final (heap : value) is group-update-related to the final (heap : value) obtained with NS, and conversely.

Theorem 2. *(Equivalence NS-NNS).*

1. $\Gamma : t \Downarrow \Delta : w \Rightarrow \exists \Delta_N \in LNHeap\,.\,\exists w_N \in Val\,.\,\Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_N : w_N) \succsim_{GU} (\Delta : w)$.
2. $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow \exists \Delta \in LNHeap\,.\,\exists w \in Val\,.\,\Gamma : t \Downarrow \Delta : w \wedge (\Delta_N : w_N) \succsim_{GU} (\Delta : w)$.

Likewise to Theorem 1, this result cannot be proved directly by rule induction and a generalization is needed. At present we are working on the proof of this generalization and other intermediate results. This may lead to slight modifications of the relations defined in this section.

Some of the problems that we have found in the proof of the generalization of the theorem are due to the fact that semantics rules for variables are different. Working with NS the variable that is being evaluated is removed from the heap while it remains there when applying NNS. In order to apply rule induction we have to remove this variable from the heap in NNS, but we should be careful because several variables can be related with this one, and all of them must be removed not to lose the group relation between heaps. We also find a problem with the update relation and we probably may to add another intermediate semantics where the variable to be evaluated is not removed from the heap but it is updated when the value is obtained.

5 Conclusions and Future Work

The variations introduced by Launchbury in its alternative natural semantics (ANS) do affect two rules: The variable rule (no update / no blackholes) and the application rule (indirections). We have defined two intermediate semantics to deal separately with the effects of each modification: NNS (without update / without blackholes) and INS (with indirections). Subsequently, we have studied the differences between the heaps obtained by the reduction systems corresponding to each semantics.

To begin with we have compared NNS with ANS, that is, substitution vs. indirections. To this purpose we have defined a preorder $\succsim_I$ expressing that a heap can be transformed into another by eliminating indirections. Furthermore, the relation $\succsim_I$ has been extended to (heap : terms) pairs, expressing that two terms can be considered equivalent when they have the same structure and their free variables (only those defined in the context of the corresponding heap) are the same except for some indirections. We have used this extended relation to establish the equivalence between the NNS and the ANS (Theorem 1).

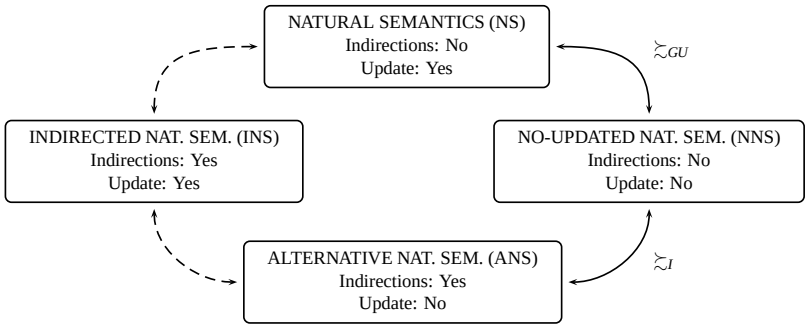


Figure 6: The relations between the semantics

Thereafter we have compared NS with NNS, that is, update vs. no update. The absence of update implies the duplication of evaluation work, that leads to the generation of duplicated bindings. These duplicated bindings come from the evaluation of `let`-declarations, so that they form *groups*. Therefore, we have defined a *group-update*-relation $\succsim_{GU}$ that relates two heaps whenever the first can be transformed into the second by first eliminating duplicated groups of bindings, and then updating the bindings. We have extended $\succsim_{GU}$ for (heap : terms) to formulate an equivalence theorem for NS and NNS (Theorem 2). This closes the path from NS to ANS, and justifies their equivalence. A schema of the semantics and their relations is shown in Figure 6.

As we have mentioned before, we are still working on the proof of Theorem 2. When done we would like to complete the picture by comparing NS with INS, and then INS with ANS. For the first step, we have to define a preorder similar to $\succsim_{I}$, but taking into account that extra indirections may now be updated, thus leading to "redundant" bindings. For the second step, some variation of the group-update-relation will be needed. Dashed lines in Figure 6 indicate this future work.

We have chosen to use a locally nameless representation to avoid the problems with α-equivalence, and we have introduced cofinite quantification (in the style of [1]) in the evaluation rules that introduce fresh names, namely the rule for local declarations (LNLET) and for the alternative application (ALNAPP). Moreover, this representation is more amenable to formalization in proof assistants. In fact we have started to implement the semantic rules given in Section 2.2 using Coq [4], with the intention of obtaining a formal checking of our proofs.

Acknowledgements: This work is partially supported by the projects TIN2012-39391-C04-04 and S2009/TIC-1465.

References

[1] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack & S. Weirich (2008): *Engineering formal metatheory*. In: *ACM Symposium on Principles of Programming Languages, POPL'08*, ACM Press, pp. 3–15.

[2] C. Baker-Finch, D. King & P. W. Trinder (2000): *An Operational Semantics for Parallel Lazy Evaluation*. In: *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal, Canada, pp. 162–173.

[3] H. P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. *Studies in Logic and the Foundations of Mathematics* 103, North-Holland.

[4] Y. Bertot (2006): *Coq in a Hurry*. *CoRR* abs/cs/0603118. Available at `http://arxiv.org/abs/cs/0603118`.

[5] J. Breitner (2013): *The Correctness of Launchbury's Natural Semantics for Lazy Evaluation*. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Launchbury.shtml`, Formal proof development, Amended version May 2014.

[6] N. G. de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes Mathematicae* 75(5), pp. 381–392.

[7] A. Charguéraud (2011): *The Locally Nameless Representation*. *Journal of Automated Reasoning*, pp. 1–46.

[8] M. van Eekelen & M. de Mol (2007): *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101. Radboud University Nijmegen.

[9] J. Launchbury (1993): *A Natural Semantics for Lazy Evaluation*. In: *ACM Symp. on Principles of Programming Languages, POPL'93*, ACM Press, pp. 144–154.

[10] K. Nakata & M. Hasegawa (2009): *Small-step and big-step semantics for call-by-need*. *Journal of Functional Programming* 19(6), pp. 699–722.

[11] T. Nipkow, L. C. Paulson & M. Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[12] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2010): *Call-by-need, call-by-name, and natural semantics*. Technical Report UU-CS-2010-020, Department of Information and Computing Sciences, Utrech University.

[13] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2010): *Trends in Functional Programming*, chapter An Operational Semantics for Distributed Lazy Evaluation, pp. 65–80. 10, Intellect.

[14] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2012): *A locally nameless representation for a natural semantics for lazy evaluation*. Technical Report 01/12, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid. Http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-1-12.pdf.

[15] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2012): *A Locally Nameless Representation for a Natural Semantics for Lazy Evaluation*. In: *Theoretical Aspects of Computing ICTAC 2012*, LNCS 7521, Springer, p. 105119.

[16] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2013): *The role of indirections in lazy natural semantics (extended version)*. Technical Report 13/13, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid. Http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/TR-13-13.pdf.

[17] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2014): *The role of indirections in lazy natural semantics*. In: *Proceedings of Ershov Informatics Conference (the PSI Conference Series, 9th edition)*. To appear.

[18] P. Sestoft (1997): *Deriving a lazy abstract machine*. *Journal of Functional Programming* 7(3), pp. 231–264.

[19] C. Urban, S. Berghofer & M. Norrish (2007): *Barendregt's Variable Convention in Rule Inductions*. In: *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, LNCS 4603, Springer-Verlag, pp. 35–50.

[20] C. Urban & C. Kaliszyk (2012): *General Bindings and Alpha-Equivalence in Nominal Isabelle*. *Logical Methods in Computer Science* 8(2:14), pp. 1–35.

Electronic Communications of the EASST
Volume X (2012)

Proceedings of the XII Spanish Conference on Progamming and Computer Languages (PROLE'12)

A formalization in Coq of Launchbury's natural semantics for lazy evaluation

Lidia Sánchez-Gil Mercedes Hidalgo-Herrero Yolanda Ortega-Mallén

15 pages

Guest Editors: María-del-Mar Gallardo
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A formalization in Coq of Launchbury's natural semantics for lazy evaluation*

Lidia Sánchez-Gil[1] **Mercedes Hidalgo-Herrero**[2] **Yolanda Ortega-Mallén**[3]

[1,3]Dpt. Sistemas Informáticos y Computación, Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain, lidiasg@mat.ucm.es, yolanda@sip.ucm.es
[2]Dpt. Didáctica de las Matemáticas, Facultad de Educación, Universidad Complutense de Madrid, Spain, mhidalgo@edu.ucm.es

Abstract: We are working on the implementation of Launchbury's semantics for lazy evaluation in the proof assistant Coq. We use a locally nameless representation where names are reserved for free variables, while bound variable names are replaced by indices. This avoids the need of α-conversion and Barendregt's variable convention, and facilitates the formalization in Coq. Simultaneous recursive local declarations in the calculus require the management of multibinders and the use of mutually inductive types.

Keywords: Formalization, locally nameless representation, proof assistant, Coq, natural semantics, lazy evaluation.

1 Motivation

Call-by-need evaluation, which avoids repeated computations, is the semantic foundation for lazy functional programming languages like Haskell or Clean. Launchbury defines in [Lau93] a natural semantics for lazy evaluation where the set of *bindings*, i.e., (variable, expression) pairs, is explicitly handled to make possible their sharing. To prove that this lazy semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury defines an alternative semantics. On the one hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach of this mechanism, applications are carried out in the alternative semantics by introducing indirections instead of by performing the β-reduction through substitution. On the other hand, the update of bindings with their computed values is an operational notion without a denotational counterpart. Thus, the alternative semantics does no longer update bindings and becomes a *call-by-name* semantics.

Alas, the proof of the equivalence between the lazy semantics and its alternative version is detailed nowhere, and a simple induction turns out to be insufficient. Intuitively, both reduction systems should produce the same results, but this cannot be directly established. Values may contain free variables that depend on the context of evaluation, which is represented by the heap of bindings. The changes introduced by the alternative semantics do deeply affect these heaps.

* Work partially supported by the projects TIN2009-14599-C03-01 and S2009/TIC-1465.

Although indirections and "duplicated" bindings (a consequence of not updating) do not add relevant information to the context, it is awkward to prove this fact.

We intend to prove formally the equivalence between Launchbury's semantics and its alternative version. In the usual representation of the λ-calculus, i.e., with variable names for free and bound variables, terms are identifed up to α-conversion. Dealing with α-equated terms usually implies the use of Barendregt's variable convention [Bar84] to avoid the renaming of bound variables. However, the use of the variable convention in rule inductions is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [UBN07]). Looking for a system of binding more amenable to formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [Cha11]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [dB72], while free variables preserve their names. Hence, α-conversion is no longer needed and variable substitution is easily defined because there is no danger of name capture. Moreover, this representation is suitable for working with proof assistants like Coq [Ber06] or Isabelle [NPW02]. We have preferred Coq to Isabelle for our formalization because Coq allows the definition of inductive types, that are needed for the recursive local declarations, while we encountered some problems when we first tried with Isabelle. Furthermore, Coq offers dependent types and proof by reflection, that will be helpful for our theory and proof developments.

Our concern for reproducing and formalizing the proof of this equivalence is not arbitrary. Launchbury's natural semantics has been cited frequently and has inspired many further works as well as several extensions of his semantics , where the corresponding adequacy proofs have been obtained by just adapting Launchbury's proof scheme. We have extended ourselves the λ-calculus with a new expression that introduces parallelism in functional applications [SHO10].

The paper is structured as follows: In Section 2 we present the locally nameless representation of the λ-calculus extended with mutually recursive local declarations. In Section 3, we express Launchbury's semantic rules in the new style and present several properties of the reduction system that are useful for the equivalence proof.[1] In Section 4 we describe the current state of the implementation in Coq of this locally nameless translation of the syntax and the semantics. In Section 5 we comment on some related work. In the last section we explain what we have achieved so far and what remains to be done.

2 The locally nameless representation

The language described by Launchbury in [Lau93] is a *normalized* λ-calculus extended with recursive local declarations. We reproduce the restricted syntax in Figure 1.a. Normalization is achieved in two steps. First an α-conversion is carried out so that all bound variables have distinct names. In a second phase, arguments for applications are enforced to be variables. These *static* transformations simplify the definition of the reduction rules.

Next, we follow the methodology summarized in [Cha11]:

1. Define the syntax of the extended λ-calculus in the locally nameless style.
2. Define the variable opening and variable closing operations.

[1] The "paper-and-pencil" proofs of these lemmas and other auxiliary results are detailed in [SHO12].

$x \in Var$
$e \in Exp ::= \lambda x.e \mid (e\ x) \mid x \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e.$

(a) Restricted *named* syntax

$x \in Id \qquad i, j \in \mathbb{N}$
$v \in Var \quad ::= \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x$
$t \in LNExp \quad ::= v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t$

(b) Locally nameless syntax

Figure 1: Extended λ-calculus

3. Define the free variables and substitution functions, as well as the local closure predicate.
4. State and prove the properties of the operations on terms that are needed in the development to be accomplished.

2.1 Locally nameless syntax

The locally nameless (restricted) syntax is shown in Figure 1.b. *Var* stands now for the set of *variables*, where we distinguish between *bound variables* and *free variables*. The calculus includes two binding constructions: λ-abstraction and `let`-declaration. Being the latter a *multi-binder*, we follow Charguéraud [Cha11] and represent bound variables with two natural numbers: The first number is a de Bruijn index that counts how many binders (abstraction or `let`) one needs to cross to the left to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind only one variable; thus, the second number should be zero. In the following, we will represent a list like $\{t_i\}_{i=1}^{n}$ as $\overline{t}$, with length $|\overline{t}| = n$.

Example 1 *Let $e \in Exp$ be an expression in the named representation:*

$$e \equiv \lambda z.\texttt{let}\ x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2, x_3 = x\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in LNExp$ is:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{fvar}\ x\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms.

As bound variables are nameless, the first phase of Launchbury's normalization is unneeded. However, application arguments are still restricted to variables.

2.2 Variable opening and variable closing

Variable opening and *variable closing* are the main operations to manipulate locally nameless terms. We extend the definitions given by Charguéraud in [Cha11] to the `let`-declaration.[2]

[2] Multiple binders are defined in [Cha11]. Two constructions are given: One for non-recursive local declarations, and another for mutually recursive expressions. Yet both extensions are not completely developed.

$$
\begin{array}{lll}
\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) & = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases} \\
\{k \to \overline{x}\}(\texttt{fvar}\ x) & = \texttt{fvar}\ x \\
\{k \to \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \to \overline{x}\}\ t) \\
\{k \to \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \to \overline{x}\}\ t)\ (\{k \to \overline{x}\}\ v) \\
\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \to \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}\ t)
\end{array}
$$

where $\{k \to \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \to \overline{x}\}\cdot)\ \overline{t}$.

Figure 2: Variable opening

To explore the body of a binder (abstraction or `let`), the corresponding bound variables are replaced with fresh names. In the case of `abs` t the *variable opening operation* replaces in t with a (fresh) name every bound variable that refers to the outermost abstraction. Similarly, to open `let` $\overline{t}$ `in` t we provide a list of $|\overline{t}|$ distinct fresh names to replace those bound variables that occur in $\overline{t}$ and in t which refer to this particular declaration.

Variable opening is defined by means of a more general function $\{k \to \overline{x}\}t$ (Figure 2), where the number k represents the nesting level of the binder to be opened, and $\overline{x}$ is a list of pairwise-distinct identifiers in *Id*. Since the level of the outermost binder is 0, variable opening is defined as $t^{\overline{x}} = \{0 \to \overline{x}\}t$. We extend this operation to lists of terms: $\overline{t}^{\overline{x}} = \texttt{List.map}\ (\cdot^{\overline{x}})\ \overline{t}$.

The last definition and those in Figure 2 include some operations on lists. We use an ML-like notation. For instance, `List.nth` $j\ \overline{x}$ represents the $(j+1)^{th}$ element of $\overline{x}$,[3] and `List.map` $f\ \overline{t}$ applies function f to every term in $\overline{t}$. In the rest of definitions we will use similar list operations.

Example 2 *Consider the term* `abs` (`let bvar 0 1,bvar 1 0 in app` (`abs bvar 2 0`) (`bvar 0 1`)). *Hence, the body of the abstraction is:*

$$u \equiv \texttt{let bvar 0 1,}\boxed{\texttt{bvar 1 0}}\ \texttt{in app (abs}\ \boxed{\texttt{bvar 2 0}}\texttt{) (bvar 0 1)}.$$

But then in u the bound variables referring to the outermost abstraction (shown squared) point to nowhere. Therefore, we consider $u^{[x]}$ *instead of u, where*

$$
\begin{array}{lll}
u^{[x]} & = & \{0 \to x\}(\texttt{let bvar 0 1,bvar 1 0 in app (abs bvar 2 0) (bvar 0 1)}) \\
& = & \texttt{let}\ \{1 \to x\}(\texttt{bvar 0 1,bvar 1 0})\ \texttt{in}\ \{1 \to x\}(\texttt{app (abs bvar 2 0) (bvar 0 1)}) \\
& = & \texttt{let bvar 0 1,fvar}\ x\ \texttt{in app (abs}\ \{2 \to x\}(\texttt{bvar 2 0}))\ \texttt{(bvar 0 1)} \\
& = & \texttt{let bvar 0 1,fvar}\ x\ \texttt{in app (abs fvar}\ x)\ \texttt{(bvar 0 1)}
\end{array}
$$

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of t is represented by $^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct). Again, a general function $\{k \leftarrow \overline{x}\}t$ (Figure 3) is defined. Whenever `fvar` x is encountered, x is looked up in $\overline{x}$: If x occurs in position j, then the free variable is replaced by `bvar` $k\ j$, otherwise it is left unchanged. Variable closing is then defined as $^{\backslash\overline{x}}t = \{0 \leftarrow \overline{x}\}t$. Its extension to lists is $^{\backslash\overline{x}}\overline{t} = \texttt{List.map}\ (^{\backslash\overline{x}}\cdot)\ \overline{t}$.

Example 3 *We close the term obtained by opening the body of the abstraction in Example 2 (i.e., the term u):* $t \equiv$ `let bvar 0 1,fvar` x `in app` (`abs fvar` x) (`bvar 0 1`).

[3] In order to better accommodate to bound variables indices, elements in a list are numbered starting with 0.

$$
\begin{array}{lll}
\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & = \texttt{bvar}\ i\ j & \\
\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) & = \begin{cases} \texttt{bvar}\ k\ j & \text{if } \exists j : 0 \leq j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise} \end{cases} & \\
\{k \leftarrow \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}\ t) & \\
\{k \leftarrow \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \leftarrow \overline{x}\}\ t)\ (\{k \leftarrow \overline{x}\}\ v) & \\
\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}\ t) &
\end{array}
$$

where $\{k \leftarrow \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \leftarrow \overline{x}\}\cdot)\ \overline{t}$.

Figure 3: Variable closing

$$
\text{LC\_VAR}\ \frac{}{\texttt{lc}\ (\texttt{fvar}\ x)} \qquad \text{LC\_ABS}\ \frac{\forall x \notin L \subseteq Id \quad \texttt{lc}\ t^{[x]}}{\texttt{lc}\ (\texttt{abs}\ t)} \qquad \text{LC\_APP}\ \frac{\texttt{lc}\ t \quad \texttt{lc}\ v}{\texttt{lc}\ (\texttt{app}\ t\ v)}
$$

$$
\text{LC\_LET}\ \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id \quad \texttt{lc}\ [t : \overline{t}]^{\overline{x}}}{\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)} \qquad \text{LC\_LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc}\cdot)\ \overline{t}}{\texttt{lc}\ \overline{t}}
$$

Figure 4: Local closure

$$
\begin{array}{lll}
{}^{\backslash x}t & = & \{0 \leftarrow x\}(\texttt{let}\ \{\texttt{bvar}\ 0\ 1, \texttt{fvar}\ x\}\ \texttt{in}\ \texttt{app}\ (\texttt{abs}\ (\texttt{fvar}\ x))\ (\texttt{bvar}\ 0\ 1)) \\
& = & \texttt{let}\ \{1 \leftarrow x\}(\texttt{bvar}\ 0\ 1, \texttt{fvar}\ x)\ \texttt{in}\ \{1 \leftarrow x\}(\texttt{app}\ (\texttt{abs}\ \texttt{fvar}\ x)\ (\texttt{bvar}\ 0\ 1)) \\
& = & \texttt{let}\ \texttt{bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in}\ \texttt{app}\ (\texttt{abs}\ \{2 \leftarrow x\}(\texttt{fvar}\ x))\ (\texttt{bvar}\ 0\ 1) \\
& = & \texttt{let}\ \texttt{bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in}\ \texttt{app}\ (\texttt{abs}\ \texttt{bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1)
\end{array}
$$

Notice that the closed term coincides with u, although this is not always the case.

2.3 Local closure, free variables and substitution

The locally nameless syntax in Figure 1.b allows to build terms without a corresponding expression in *Exp*, e.g., the term $\texttt{abs}\ (\texttt{bvar}\ 1\ 5)$ where index 1 does not refer to a binder in the term. Well-formed terms, i.e., those that correspond to expressions in *Exp*, are called *locally closed*.

To determine if a term is locally closed one should check that every bound variable in the term has valid indices, i.e., they refer to binders in the term. An easier method is to open with fresh names every abstraction and `let`-declaration in the term to be checked, and prove that no bound variable is ever reached. This is implemented with the *local closure* predicate `lc` (Figure 4).

Observe that we use cofinite quantification (as introduced by Aydemir et al. in [ACP+08]) in the rules for the binders, i.e., abstraction and `let`. Cofinite quantification is an elegant alternative to exist-fresh conditions and provides stronger induction principles. Proofs are simplified, as it is not required to define exactly the set of fresh names (examples of this are given in [Cha11]). The rule LC-ABS establishes that an abstraction is locally closed if there exists a finite set of names L such that, for any name x not in L, the term $t^{[x]}$ is locally closed. Similarly, in the rule LC-LET we write $\overline{x}^{|\overline{t}|} \notin L$ to indicate that the list of distinct names $\overline{x}$ of length $|\overline{t}|$ are not in the finite set L. For any list $\overline{x}$ satisfying this condition, the opening of each term in the list of local declarations, $\overline{t}^{\overline{x}}$, and of the term affected by these declarations, $t^{\overline{x}}$, are locally closed. We have overloaded the predicate `lc` to work both on terms and lists of terms; later on we will overload other predicates and functions similarly. We write $[t : \overline{t}]$ for the list with head t and tail $\overline{t}$. In the following, $[\,]$ represents the empty list, $[t]$ is a unitary list, and $+\!\!+$ is the concatenation of lists.

$$\text{LCK-BVAR}\quad \frac{i < k \wedge j < \texttt{List.nth}\ i\ \overline{n}}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)} \qquad \text{LCK-APP}\quad \frac{\texttt{lc\_at}\ k\ \overline{n}\ t \qquad \texttt{lc\_at}\ k\ \overline{n}\ v}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t\ v)}$$

$$\text{LCK-FVAR}\quad \frac{}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{fvar}\ x)} \qquad \text{LCK-LET}\quad \frac{\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ [t : \overline{t}]}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LCK-ABS}\quad \frac{\texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ t}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t)} \qquad \text{LCK-LIST}\quad \frac{\texttt{List.forall}\ (\texttt{lc\_at}\ k\ \overline{n}\ \cdot)\ \overline{t}}{\texttt{lc\_at}\ k\ \overline{n}\ \overline{t}}$$

Figure 5: Local closure at level k

$$\begin{array}{ll|ll}
\texttt{fv}(\texttt{bvar}\ i\ j) & = \emptyset & (\texttt{bvar}\ i\ j)[z/y] & = \texttt{bvar}\ i\ j \\
\texttt{fv}(\texttt{fvar}\ x) & = \{x\} & (\texttt{fvar}\ x)[z/y] & = \begin{cases} \texttt{fvar}\ z & \text{if } x = y \\ \texttt{fvar}\ x & \text{if } x \neq y \end{cases} \\
\texttt{fv}(\texttt{abs}\ t) & = \texttt{fv}(t) & (\texttt{abs}\ t)[z/y] & = \texttt{abs}\ t[z/y] \\
\texttt{fv}(\texttt{app}\ t\ v) & = \texttt{fv}(t) \cup \texttt{fv}(v) & (\texttt{app}\ t\ v)[z/y] & = \texttt{app}\ t[z/y]\ v[z/y] \\
\texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{fv}(\overline{t}) \cup \texttt{fv}(t) & (\texttt{let}\ \overline{t}\ \texttt{in}\ t)[z/y] & = \texttt{let}\ \overline{t}[z/y]\ \texttt{in}\ t[z/y]
\end{array}$$

where $\texttt{fv}(\overline{t}) = \texttt{List.foldright}\ (\cdot \cup \cdot)\ \emptyset\ (\texttt{List.map}\ \texttt{fv}\ \overline{t})$
$\overline{t}[z/y] = \texttt{List.map}\ ([z/y]\cdot)\ \overline{t}$

Figure 6: Free variables and substitution

We define a new predicate that checks if indices in bound variables are valid from a given level: *t is closed at level k*, written $\texttt{lc\_at}\ k\ \overline{n}\ t$ (Figure 5), where k indicates the current binding depth. Since we are dealing with multibinders, we need to keep their size (1 in the case of an abstraction, n for a `let` with n local declarations). These sizes are collected in the list $\overline{n}$, so that $|\overline{n}|$ should be at least k. A bound variable $\texttt{bvar}\ i\ j$ is closed at level k if i is smaller than k and j is smaller than $\texttt{List.nth}\ i\ \overline{n}$. The list $\overline{n}$ is new with respect to [Cha11] because there the predicate `lc_at` is not defined for multiple binders.

The two approaches for local closure are equivalent, so that a term is locally closed if and only if it is closed at level 0: LC_IIF_LC_AT $\texttt{lc}\ t \Leftrightarrow \texttt{lc\_at}\ 0\ []\ t$.

Computing the *free variables* of a term t is very easy in the locally nameless representation, since bound and free variables are syntactically different. The set of free variables of $t \in LNExp$ is denoted as $\texttt{fv}(t)$, and it is defined in Figure 6.

A name x is said to be *fresh for a term t*, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t. Similarly for a list of distict names $\overline{x}$:

$$\frac{x \notin \texttt{fv}(t)}{\texttt{fresh}\ x\ \texttt{in}\ t} \qquad \frac{\overline{x} \notin \texttt{fv}(t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ t}$$

Substitution replaces a variable name by another name in a term. So that for $t \in LNExp$ and $z, y \in Id$, $t[z/y]$ is the term where z substitutes any occurrence of y in t (see Figure 6).

Under some conditions variable opening and closing are inverse operations. More precisely, opening a term with fresh names and closing it with the same names, produces the original term. Closing a locally closed term and then opening it with the same names gives back the term:

CLOSE_OPEN_VAR $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow {}^{\backslash \overline{x}}(t^{\overline{x}}) = t$ OPEN_CLOSE_VAR $\texttt{lc}\ t \Rightarrow ({}^{\backslash \overline{x}}t)^{\overline{x}} = t$

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \mathtt{let}\ \{x_i = e_i\}_{i=1}^n\ \mathtt{in}\ e \Downarrow \Delta : w}$$

Figure 7: Natural semantics

3 Natural semantics for lazy evaluation

The natural semantics defined by Launchbury [Lau93] follows a lazy strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : w$, that is, the expression $e \in Exp$ in the context of the heap Γ reduces to the value w in the context of the heap Δ. *Values* ($w \in Val$) are expressions in weak-head-normal-form (*whnf*). *Heaps* are partial functions from variables into expressions. Each pair (variable, expression) is called a *binding*, represented by $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings may be updated to their corresponding computed values. The semantic rules are shown in Figure 7. The normalization of the λ-calculus, as mentioned in Section 2, simplifies the definition of the rules, although a renaming is still needed ($\hat{w}$ in VAR) to avoid name clashing. This renaming is justified by Barendregt's variable convention [Bar84].

Example 4 *Without the renaming in rule* VAR *heaps may end up binding a name more than once. Take for instance the evaluation of the expression*

$$e \equiv \mathtt{let}\ x_1 = \lambda y.(\mathtt{let}\ z = \lambda v.y\ \mathtt{in}\ y), x_2 = (x_1\ x_3), x_3 = (x_1\ x_4), x_4 = \lambda s.s\ \mathtt{in}\ x_2$$

in the context of the empty heap. The evaluation of e implies the evaluation of x_2, and then the evaluation of $(x_1 x_3)$. This application leads to the addition of $z \mapsto \lambda v.x_3$ to the heap. Eventually, the evaluation of x_3 implies the evaluation of $(x_1 x_4)$. Without a renaming of values, variable z is added again to the heap, now bound to $\lambda v.x_4$.

Theorem 1 in [Lau93] states that "every heap/term pair occurring in the proof of a reduction is *distinctly named*", but we have found that the renaming fails to ensure this property. At least, it depends on how much fresh is this renaming.

Example 5 *Let us evaluate in the context of the empty heap the following expression:*

$$e \equiv \mathtt{let}\ x_1 = (x_2\ x_3), x_2 = \lambda z.\mathtt{let}\ y = \lambda t.t\ \mathtt{in}\ y, x_3 = \lambda s.s\ \mathtt{in}\ x_1$$

$$\begin{array}{l}
\{\,\} : e \\
\textsc{Let}\ \Big|\ \{x_1 \mapsto (x_2\ x_3), x_2 \mapsto \lambda z.\mathtt{let}\ y = \lambda t.t\ \mathtt{in}\ y, x_3 \mapsto \lambda s.s\} : x_1 \\
\quad \textsc{Var}\ \Big|\ \{x_2 \mapsto \lambda z.\mathtt{let}\ y = \lambda t.t\ \mathtt{in}\ y, x_3 \mapsto \lambda s.s\} : (x_2\ x_3) \\
\quad\quad \textsc{App}\ \Big|\ \{x_2 \mapsto \lambda z.\mathtt{let}\ y = \lambda t.t\ \mathtt{in}\ y, x_3 \mapsto \lambda s.s\} : x_2 \\
\quad\quad\quad \textsc{Var}\ \Big|\ \{x_3 \mapsto \lambda s.s\} : \lambda z.\mathtt{let}\ y = \lambda t.t\ \mathtt{in}\ y \\
\quad\quad\quad\quad \textsc{Lam}\ \Big|\ \\
\quad\quad\quad\quad \{x_3 \mapsto \lambda s.s\} : \boxed{\lambda z.\mathtt{let}\ y = \lambda t.t\ \mathtt{in}\ y}
\end{array}$$

At this point the VAR*-rule requires to rename the value that has been obtained (which is highlighted in the square). Notice that x_1 is fresh in the actual heap/term pair, and thus could be*

chosen to rename y. This would lead later in the derivation to introduce x_1 twice in the heap. The solution is to consider the condition of freshness in the whole derivation. This notion has not been formally defined by Launchbury.

3.1 Locally nameless heaps

Before translating the semantic rules in Figure 7 to the locally nameless representation defined in Section 2, we have to show how *bindings* and *heaps* are represented in this notation. Bindings associate expressions to free variables, hence bindings are now pairs $(\texttt{fvar}\ x, t)$ with $x \in Id$ and $t \in LNExp$. To simplify, we just write $x \mapsto t$. In the following, we represent a heap $\{x_i \mapsto t_i\}_{i=1}^{n}$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names that are bound in the heap:

$$\texttt{dom}(\emptyset) = \emptyset \qquad\qquad \texttt{dom}(\Gamma, x \mapsto t) = \texttt{dom}(\Gamma) \cup \{x\}$$

In a well-formed heap names are defined at most once and terms are locally closed. The predicate $\texttt{ok}$ expresses that a heap is well-formed:

$$\text{OK-EMPTY}\ \frac{}{\texttt{ok}\ \emptyset} \qquad\qquad \text{OK-CONS}\ \frac{\texttt{ok}\ \Gamma \qquad x \notin \texttt{dom}(\Gamma) \qquad \texttt{lc}\ t}{\texttt{ok}\ (\Gamma, x \mapsto t)}$$

The function $\texttt{names}$ returns the set of names that appear in a heap, i.e., the names occurring in the domain or in the right-hand side terms:

$$\texttt{names}(\emptyset) = \emptyset \qquad\qquad \texttt{names}(\Gamma, x \mapsto t) = \texttt{names}(\Gamma) \cup \{x\} \cup \texttt{fv}(t)$$

This definition can be extended to (heap: term) pairs: $\texttt{names}(\Gamma : t) = \texttt{names}(\Gamma) \cup \texttt{fv}(t)$
We also extend the freshness predicate:

$$\frac{\overline{x} \notin \texttt{names}(\Gamma : t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : t)}$$

Substitution of variable names is extended to heaps as follows:

$$\emptyset[z/y] = \emptyset \qquad\qquad (\Gamma, x \mapsto t)[z/y] = (\Gamma[z/y], x[z/y] \mapsto t[z/y])$$
$$\textit{where}\ x[z/y] = \begin{cases} z & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

3.2 Locally nameless semantics

Once the locally nameless syntax and the corresponding operations, functions and predicates have been defined, three steps are sufficient to translate an inductive definition on λ-terms from the named representation into the locally nameless notation (as it is explained in [Cha11]):

1. Replace the named binders, i.e., abstractions and let-declarations, with nameless binders by opening the bodies.
2. Cofinitely quantify the names introduced for variable opening.
3. Add premises to inductive rules in order to ensure that inductive judgements are restricted to locally closed terms.

$$\text{LNLAM}\quad \frac{\{\texttt{ok}\ \Gamma\} \qquad \{\texttt{lc}\ (\texttt{abs}\ t)\}}{\Gamma : \texttt{abs}\ t \Downarrow \Gamma : \texttt{abs}\ t}$$

$$\text{LNVAR}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \cup \texttt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : (\texttt{fvar}\ x) \Downarrow (\Delta, x \mapsto w) : w}$$

$$\text{LNAPP}\quad \frac{\Gamma : t \Downarrow \Theta : \texttt{abs}\ u \qquad \Theta : u^{[x]} \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin \texttt{dom}(\Delta)\}}{\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x) \Downarrow \Delta : w}$$

$$\text{LNLET}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \qquad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Figure 8: Locally nameless natural semantics

We apply these steps to the inductive rules for the lazy natural semantics given in Figure 7. These rules produce judgements involving λ-terms as well as heaps. Hence, we also add premises that ensure that inductive judgements are restricted to well-formed heaps. The rules using the locally nameless representation are shown in Figure 8. For clarity, in the rules we put in braces the side-conditions to distinguish them better from the judgements.

The main difference with the rules in Figure 7 is the rule LNLET. To evaluate $\texttt{let}\ \overline{t}\ \texttt{in}\ t$ the local terms in $\overline{t}$ are introduced in the heap, so that the body t is evaluated in this new context. Fresh names $\overline{x}$ are needed to open the local terms and the body. The evaluation of $t^{\overline{x}}$ produces a final heap and a value. Both are dependent on the names chosen for the local variables. The domain of the final heap consists of the local names $\overline{x}$ and the rest of names, say $\overline{z}$. The rule LNLET is cofinite quantified. As it is explained in [Cha11], the advantage of the cofinite rules over existential and universal ones is that the freshness side-conditions are not explicit. In our case, the freshness condition for $\overline{x}$ is *hidden* in the finite set L, which includes the names that should be avoided during the reduction. The novelty of our cofinite rule, compared with the ones appearing in [ACP+08] and [Cha11] (that are similar to the cofinite rules for the predicate $\texttt{lc}$ in Figure 4), is that the names introduced in the (infinite) premises do appear in the conclusion too. Hence, in the conclusion of the rule LNLET we can replace the names $\overline{x}$ by any list $\overline{y}$ not in L.

The problem with explicit freshness conditions is that they are associated just to rule instances, while they should apply to the whole reduction proof. Take for instance the rule LNVAR. In the premise the binding $x \mapsto t$ does no longer belong to the heap. Therefore, a valid reduction for this premise may chose x as fresh (this corresponds to the problem shown in Example 5). We avoid this situation by requiring that x remains undefined in the final heap too. By contrast to the rule VAR in Figure 7, no renaming of the final value, that is w, is needed.

The new side-condition of rule LNAPP deserves an explanation. Suppose that x is undefined in the initial heap Γ, then we must avoid that x is chosen as a fresh name during the evaluation of t. For this reason we require that x is defined in the final heap Δ only if x was already defined in Γ. Notice how the body of the abstraction, that is u, is open with the name x. This is equivalent to the substitution of x for y in the body of the abstraction $\lambda y.e'$ (see rule APP in Figure 7).

A *regularity* lemma ensures that the judgements produced by this reduction system involve only well-formed heaps and locally closed terms.

REGULARITY $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \texttt{ok}\ \Gamma \wedge \texttt{lc}\ t \wedge \texttt{ok}\ \Delta \wedge \texttt{lc}\ w.$

Similarly, Theorem 1 in [Lau93] ensures that the property of being *distinctly named* is preserved by the rules in Figure 7. However, as shown in Example 5, the correctness of this result requires that freshness is relative to whole reduction proofs instead to the scope of rules.

Names defined in a context heap remain defined after the evaluation of any term in that context:

DEF_NOT_LOST $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta)$

Furthermore, fresh names are introduced only by the rule LNLET and, by the previous result, they remain bound in the final (heap: value) pair. Hence, any free variable appearing in a final (heap: value) pair is undefined only if the variable already occurs in the initial (heap: term) pair.

ADD_VARS $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow (x \in \mathtt{names}(\Delta : w) \Rightarrow (x \in \mathtt{dom}(\Delta) \vee x \in \mathtt{names}(\Gamma : t)))$

A *renaming* lemma ensures that the evaluation of a term is independent of the fresh names chosen in the reduction process. Moreover, any name in the context can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reduction proofs for (heap: term) pairs are unique up to renaming of the variables defined in the context heap.

RENAMING $\quad \Gamma : t \Downarrow \Delta : w \wedge \mathtt{fresh}\, y\, \mathtt{in}\, (\Gamma : t) \wedge \mathtt{fresh}\, y\, \mathtt{in}\, (\Delta : w) \Rightarrow \Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$

In addition, the renaming lemma permits to prove an *introduction* lemma for the cofinite rule LNLET which establishes that the corresponding existential rule is admissible too.

LET_INTRO $\quad (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \wedge \mathtt{fresh}\, \overline{x}\, \mathtt{in}\, (\Gamma : \mathtt{let}\, \overline{t}\, \mathtt{in}\, t)$
$\Rightarrow \Gamma : \mathtt{let}\, \overline{t}\, \mathtt{in}\, t \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}}$

This result, together with the renaming lemma, justifies that our rule LNLET is equivalent to Launchbury's rule LET used with normalized terms.

4 Implementation in Coq

We are currently extending the work of Charguéraud [Cha11] to adapt it to our calculus and later on, formalize Launchbury's semantics. In http://www.charguéraud.org/softs/ln/ we can find the Coq library TLC which provides the representation for variables and finite sets of variables as well as the tactics that are needed to work with the locally nameless representation. The main difference between the syntax defined in [Cha11] and ours are the local definitions introduced by the `let`-terms. Although we required a restricted syntax for the application (see Figure 1.b), for the moment we have dropped this restriction in order to avoid some problems related with the induction principle.

Our first attempt was to define terms as follows:

```
Inductive trm : Set := | trm_bvar : nat → nat → trm
                       | trm_fvar : var → trm
                       | trm_abs : trm → trm
                       | trm_app : trm → trm → trm
                       | trm_let : (list trm) → trm → trm.
```

Unfortunately, when checking the induction principle defined by Coq, it does not go inside the list of terms that occurs in a local declaration. As explained by Bertot and Castéran in [BC04] we

have to redefine the induction principle that we want to use. To reach this objective we redefine the terms as:

```
Inductive trm : Set := | trm_bvar : nat → nat → trm
                       | trm_fvar : var → trm
                       | trm_abs : trm → trm
                       | trm_app : trm → trm → trm
                       | trm_let : L_trm → trm → trm
          with L_trm := | nil_Ltrm : L_trm
                       | cons_Ltrm : trm → L_trm → L_trm.
```

and afterward we give the following induction scheme:

```
Scheme   trm_ind2   := Induction for trm Sort Prop
  with   L_trm_ind2 := Induction for L_trm Sort Prop.
```

We have extended the operations for variable opening and closing. To achieve this he have completed the recursive functions open and close at level *k*. As a sample, we show here the opening at level *k*. Observe how the definition of this function reproduces the two-layers structure (terms and list of terms):

```
Fixpoint open_rec (k : nat) (vs : list var) (t : trm) {struct t} : trm :=
  match t with
   | t_bvar i j    ⇒ if (andb (beq_nat k i) (blt_nat j (length vs)))
                     then (t_fvar(nth j vs a)) else (t_bvar i j)
   | t_fvarx       ⇒ t_fvarx
   | t_abs t1      ⇒ t_abs (open_rec (S k) vs t1)
   | t_app t1 t2   ⇒ t_app (open_rec k vs t1) (open_rec k vs t2)
   | t_let ts t    ⇒ t_let (open_L_rec (S k) vs ts) (open_rec (S k) vs t)
  end
with open_L_rec (k : nat) (vs : list var) (ts : L_trm) {struct ts} : L_trm :=
  match ts with
   | nil_Lt         ⇒ nil_Lt
   | cons_Lt t ts   ⇒ cons_Lt (open_rec k vs t) (open_L_rec k vs ts)
end.
```

The function `beq_nat` returns a boolean expression indicating if two given numbers are equal or not. Analogously, the function `blt_nat` indicates if a natural number is less or equal than the second one. As expected, `length` returns the length of a list and *nth j v s a* returns the j-nth element of the list vs, where *a* is the default value for an index out of range. Notice that in our definition this default value cannot be returned due to the restrictions of the if-then-else expression.

Now the definition of the opening operation is:

```
Definition open t vs := open_rec 0 vs t.
```

Example 6 *We can define the body of the abstraction given in Example 2 as:*

```
Definition body := t_let   (cons_Lt (t_bvar 0 1) (cons_Lt (t_bvar 1 0) (nil_Lt)))
                           (t_app (t_abs (t_bvar 2 0)) (t_bvar 0 1)).
```

In the following lemma we prove that the opening of this term (called `body`*) with a variable x coincides with the result given previously in the example:*

```
Lemma demo_open : open body (cons x nil) = t_let   (cons_Lt (t_bvar 0 1) (cons_Lt (t_fvar x) nil_Lt))
                                                   (t_app (t_abs (t_fvarx)) (t_bvar 0 1)).
Proof.
unfold body. unfold open. unfold open_rec. case_if. case_if. unfold nth. case_if. auto.
Qed.
```

With the `unfold` *tactic the term* `body`*, and the functions* `open_rec` *and* `nth` *are rewritten with their definitions. The tactic* `case_if` *is used to evaluate* `if` *statements. The tactic* `auto` *ends the proof automatically by applying several tactics, including* `reflexivity` *(which considers that two syntactically equal expressions are indeed equal).*

At present we are working on the extension of the locally closed predicate, which involves the cofinite quantification over a list of identifiers (instead of only one identifier). We are working on the definition of heaps and some of its predicates as well. With these definitions we will be able to prove some properties over terms and heaps.

5 Related work

In order to avoid α-conversion, we first considered a nameless representation like the de Bruijn notation [dB72], where variable names are replaced by natural numbers. But this notation has several drawbacks. First of all, the de Bruijn representation is hard to read for humans. Even if we intend to check our results with a proof assistant, human readability helps intuition. Moreover, the de Bruijn notation does not have a good way to handle free variables, which are represented by indices, alike to bound variables. This is a serious weakness for our application. Launchbury's semantics uses context heaps that collect the bindings for the free variables that may occur in the term under evaluation. Any change in the domain of a heap, i.e., adding or deleting a binding, would lead to a shifting of the indices, thus complicating the statement and proof of results. Therefore, we prefer the more manageable locally nameless representation, where bound variable names are replaced by indices but free variables keep their names. This mixed notation combines the advantages of both named and nameless representations. On the one hand, α-conversion is avoided all the same. On the other hand, terms stay readable and easy to manipulate.

There exists in the literature different proposals for a locally nameless representation, and many works using these representations. Charguéraud offers in [Cha11] a brief survey on these works, that we recommend to the interested reader.

Launchbury implicitly assumes Barendregt's variable convention [Bar84] twice in [Lau93]. First when he defines his operational semantics only for normalized λ-terms (i.e., every binder in a term binds a distinct name, which is also distinct from any free variable); and second, when he requires a (fresh) renaming of the values in the rule VAR (Figure 7). Urban, Berghofer and Norrish propose in [UBN07] a method to strengthen an induction principle (corresponding to some inductive relation), so that the variable convention comes already built in the principle. Unfortunately, we cannot apply these ideas to Launchbury's semantics, because the semantic rules do not satisfy the conditions that guarantee the *variable convention compatibility*, as described

in [UBN07]. In fact, as we have already pointed out, Launchbury's Theorem 1 (in [Lau93]) is correct only if the renaming required in each application of the rule VAR is fresh in the whole reduction proof. Therefore, we cannot use directly Urban's nominal package for Isabelle/HOL [Urb08] (including the recent extensions for general bindings described in [UK10]).

Nevertheless, Urban et al. achieve the "inclusion" of the variable convention in an induction principle by adding to each induction rule a side condition which expresses that the set of *bound* variables (those that appear in a binding position in the rule) are fresh in some *induction context* ([UBN07]). Furthermore, this context is required to be finitely supported. This is closely related to the cofinite quantification that we have used for the rule LNLET in Figure 8. Besides, a condition to ensure the variable convention compatibility is the *equivariance* of the functions and predicates occurring in the induction rules. Equivariance is a notion from nominal logic [Pit03]. A relation is equivariant if it is preserved by permutation of names. Although we have not proven that the reduction relation defined by the rules in Figure 8 is equivariant, our *renaming lemma* establishes a similar result, that is, the reduction relation is preserved by (fresh) renaming.

Nowadays there is a wide range of proof checkers and theorem provers. Wiedijk has compiled in http://www.cs.ru.nl/ freek/digimath/ a large list of these computer systems, classified by a number of categories. Focusing in proof assistants, Barendregt and Geuvers give in [BG01] an interesting discussion and comparison of the most widely-known proof assistants, that it is still effective. We also recommend the lecture of the paper by Geuvers [Geu09] on the main concepts and history of proof assistants. We have already commented in the introduction our reasons for using Coq [Ber06], that is, dependent and inductive types, tactics, and proof by reflection.

6 Present and future

We have used a more modern approach to binding, i.e., a locally nameless representation for the λ-calculus extended with mutually recursive local declarations. With this representation the reduction rule for local declarations implies the introduction of fresh names. We have used neither an existential nor a universal rule for this case. Instead, we have opted for a cofinite rule as introduced by Aydemir et al. in [ACP+08]. Freshness conditions are usually considered in each rule individually. Nevertheless, this technique produces name clashing when considering whole reduction proofs. A solution might be to decorate judgements with the set of forbidden names and indicate how to modify this set during the reduction process (this approach has been taken by Sestoft in [Ses97]). However, this could be too restrictive in many occasions. Besides, existential rules are not easy to deal with because each reduction is obtained just for one specific list of names. If any of the names in this list causes a name clashing with other reduction proofs, then it is cumbersome to demonstrate that an alternative reduction for a fresh list does exist. Cofinite quantification solves this problem because in a single step reductions are guaranteed for an infinite number of lists of names. Nonetheless, our (LET_INTRO) lemma guarantees that a more conventional exists-fresh rule is correct in our reduction system too.

The cofinite quantification that we have used in our semantic rules is more complex than those in [ACP+08] and [Cha11]. The cofinite rule LNLET in Figure 8 introduces quantified variables in the conclusion as well, as the latter depends on the chosen names.

Compared to Launchbury's semantic rules, our locally nameless rules include several extra

side-conditions. Some of these require that heaps and terms are well-formed (e.g., in LNLAM). Others express restrictions on the choice of fresh names, so that together with the cofinite quantification, fix the problem with the renaming in rule VAR that we have shown in Example 5.

Our locally nameless semantics satisfies a *regularity lemma* which ensures that every term and heap involved in a reduction proof is well-formed, and a *renaming lemma* which indicates that the choice of names (free variables) is irrelevant as long as they are fresh enough. A heap may be seen as a multiple binder. Actually, the names defined (bound) in a heap can be replaced by other names, provided that terms keep their meaning in the context represented by the heap. Our renaming lemma ensures that whenever a heap is renamed with fresh names, reduction proofs are preserved. This renaming lemma is essential in rule induction proofs for some properties of the reduction system. More concretely, when one combines several reduction proofs coming from two or more premises in a reduction rule (for instance, in rule LNAPP in Figure 8).

So far, our contributions comprises the following:

1. A locally nameless representation of a λ-calculus with recursive local declarations;
2. A locally nameless version of Launchbury's natural semantics for lazy evaluation;
3. A new version of cofinite rules where the variables quantified in the premises do appear in the conclusion too;
4. A way to guarantee Barendregt's variable convention by redefining Launchbury's semantic rules with cofinite quantification and extra side-conditions;
5. A set of interesting properties of our reduction system, including the regularity, the introduction and the renaming lemmas; and
6. An *ongoing* implementation in Coq of Launchbury's semantics.

In the future we will use the implementation in Coq to prove formally the equivalence of Launchbury's natural semantics with the alternative version given also in [Lau93]. As we mentioned in Section 1, this alternative version differs from the original one in the introduction of indirections during β-reduction and the elimination of updates. We have already started with the definition (using the locally nameless representation) of two intermediate semantics, one introducing indirections and the other without updates. We are investigating equivalence relations between heaps obtained by each semantics, which makes able to prove the equivalence of the original natural semantics and the alternative semantics through these intermediate semantics.

References

[ACP+08] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages (POPL'08)*. Pp. 3–15. ACM Press, 2008.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, 1984.

[BC04] Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq' Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[Ber06] Y. Bertot. Coq in a Hurry. *CoRR* abs/cs/0603118, 2006.

[BG01] H. Barendregt, H. Geuvers. Proof-assistants using dependent type systems. In Robinson and Voronkov (eds.), *Handbook of automated reasoning*. Pp. 1149–1238. Elsevier Science Publishers B. V., 2001.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75(5):381–392, 1972.

[Cha11] A. Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, pp. 1–46, 2011.

[Geu09] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana* 34(1):3–25, 2009.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *ACM Symposium on Principles of Programming Languages (POPL'93)*. Pp. 144–154. ACM Press, 1993.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation* 186(2):165–193, 2003.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming* 7(3):231–264, 1997.

[SHO10] L. Sánchez-Gil, M. Hidalgo-Herrero, Y. Ortega-Mallén. *Trends in Functional Programming*. Volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pp. 65–80. Intellect, 2010.

[SHO12] L. Sánchez-Gil, M. Hidalgo-Herrero, Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. Technical report 01/12, Dpt. Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2012. http://maude.sip.ucm.es/eden-semantics/.

[UBN07] C. Urban, S. Berghofer, M. Norrish. Barendregt's Variable Convention in Rule Inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*. Pp. 35–50. Springer-Verlag, 2007.

[UK10] C. Urban, C. Kaliszyk. General Bindings and Alpha-Equivalence in Nominal Isabelle. In *Proceedings of the 20th European Symposium on Programming*. Pp. 480–500. Springer-Verlag, 2011.

[Urb08] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automatic Reasoning* 40(4):327–356, 2008.